UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Adam KHAYAM

A Meta-Approach to Describe Effectful and Distributed Semantics

Thèse présentée et soutenue à Salle Métivier (C024), INRIA Rennes, le 30/11/2022
Unité de recherche : CELTIQUE/EPICURE

Rapporteurs avant soutenance :

Catherine DUBOIS	Professeure, Ecole Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise
Sukyoung RYU	Professeure, Korea Advanced Institute of Science and Technology

Composition du Jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du jury doit être revue pour s'assurer qu'elle est conforme et devra être répercutée sur la couverture de thèse

Président :	Sandrine BLAZY	Professeure Université de Rennes 1
Examinateurs :	Xavier RIVAL	Directeur de Recherche, INRIA Paris
	Arthur CHARGUÉRAUD	Chargé de Recherche, INRIA Nancy
Dir. de thèse :	Alan SCHMITT	Directeur de Recherche, INRIA Rennes
Co-dir. de thèse :	Tamara REZK	Directrice de Recherche, INRIA Sophia Antipolis

Acknowledgement

Another experience has finished, and I am thankful to the universe for that. It seems yesterday that I started my doctoral path, and now it is its end.

First, I would like to thank the reviewers that have carefully read my thesis and given insightful comments. It is such a heavy document, so I can only thank you for your time reading and commenting on it. In the same way, I would like to thank the rest of the jury: time is valuable, and dedicating your time to me means a lot.

Now let's talk about these three years. I had a wonderful time in Rennes, even during the pandemic. One of the pillars of this period was my team (Celtique/Epicure). I thank each of you, I loved everything, but I am a fan of the small things. For example, at the pandemic's beginning, you organized "la pause café" to chat each afternoon. It was a sparkle of normality. I have a lot of respect for you, and I could not have been luckier. The Celtique/Epicure team is amazing.

A special mention goes to my friends in Rennes. We had a wonderful time during a crazy period. We were a stronghold to each other, and we made that crisis one of the best parts of our lives. I wanted to cite each one of you and your papers. A material way to thank you individually, but suddenly I realized you already have too many citations :p.

I want to thank also my colleagues in Sophia Antipolis, my friends in Nice, and the people that often visit from Paris. I always found a great environment there and felt welcomed by both the teams on the floor, INDES and STAMP.

Then, let's talk about my parents in academia, Alan and Tamara, ladies first. Thank you for hiring me; I hope I did not disappoint you. I learned a lot from you, and I am always thankful for your belief in me. Alan, it was quite an adventure. I came to you as a profane, and you had the patience and ability to introduce me to the topic. You helped me a lot with your knowledge and your personality. Whenever I was insecure about what to do, I knew that I had you covering my back.
I am lucky. I couldn't have hoped for a better academic family.

Let's go to my biological family. I thank you for your patience, mostly when I don't pick up the phone and answer back. I love you!

Last but not least: Moudy, nothing would have started without you. Thank you,

brother!

UNIVERSITÉ DE RENNES 1

Titre : Une Méta-Approche pour Décrire des Sémantiques avec Effets et Distribuée

Mot clés : Sémantique Squelettique ; Langage de Spécification ; ECMAScript ; Monades ; Systèmes Distribués ; Internet des Objets

Résumé : L'étude de la sémantique des langages de programmation est un domaine de l'informatique visant à représenter formellement le comportement de programmes. L'état de l'art a beaucoup progressé au cours des quatre dernières décennies en proposant de plus en plus de cadres adaptés à la réalisation de ces études. S'il existe de nombreux styles, dont on connaît les caractéristiques positives et les limites, il reste encore beaucoup à faire en termes d'outils adaptés à la description et à l'étude de la sémantique des langages de programmation.

Cette thèse s'inscrit dans ce contexte, en proposant une méthodologie pour pouvoir produire un objet représentant le comportement d'un langage de programmation, en fournissant tout ce qui est nécessaire pour pouvoir les étudier. La contribution de la thèse consiste à fournir une méthodologie pour écrire une sémantique concise et visuellement proche d'une spécification. Pour ce faire, nous utilisons un certain nombre de constructions algébriques qui, associées à notre approche purement fonctionnelle, conduisent à une formalisation claire, concise et facile à maintenir. Nous avons appliqué cette technique à deux études de cas, la modélisation de la spécification JavaScript, ECMAScript, et la représentation d'un modèle formel pour décrire la sémantique d'orchestration représentant le comportement des applications de l'internet des Objets. Ces travaux donnent une idée claire du potentiel et de l'expressivité de notre cadre formel, appelé sémantique squelettique [8, 41].

Ce manuscrit est à la fois une introduction à l'utilisation de la sémantique squelettique et son application aux langages de programmation, même quand ceux-ci ont une spécification complexe et de taille conséquente. C'est également une étude de la manière d'exploiter les caractéristiques d'un tel outil pour produire des formalisations qui peuvent étre claires et lisibles par l'homme. En substance, il s'agit de faire d'une représentation mathématique d'un langage, qui s'adresse à une catégorie spécifique de chercheurs, un support précieux pour les programmeurs en vue d'implémentations réelles.

UNIVERSITÉ DE RENNES 1

Title: A Meta-Approach to Describe Effectful and Distributed Semantics

Keywords: Skeletal Semantics; Specification Language; ECMAScript; Monads; Distributed Systems; Internet of Things

Abstract: The study and production of programming language semantics is a computer science field aiming to represent these objects' behavior formally. The State-of-the-art presents different styles and technologies for working on semantics. Technologies have progressed a lot during the last four decades by proposing more and more frameworks adapted to carry out these studies. On the one hand, semantic styles have well-known positive characteristics and limitations. On the other hand, semantic tools are an active research topic.

The thesis proposes a methodology for encoding semantics in a tiny functional metalanguage called Skel [41]. This language has a suite of tools for generating different artefacts for performing different types of studies. The methodology proposed in this thesis shows how to write concise semantics visually close to a programming language specification. To do so, we combined different algebraic constructs for writing precise, concise, readable, and easily maintainable formalizations. We have applied this technique to two case studies, the modeling of the JavaScript specification, ECMAScript, and the representation of a formal model to describe orchestration semantics representing the behavior of Internet of Things applications. These works give a clear idea of the potential and expressiveness of our formal framework—Skeletal Semantics [8].

This manuscript is an introduction to Skeletal Semantics, showing how to produce usable and human-readable formalizations of different types of specifications—inference-rule based and in prose. In essence, the aim is to turn a mathematical representation of a language addressed to a specific category of researchers into valuable support for programmers to build real implementations.

RÉSUMÉ EN FRANÇAIS

Tout langage de programmation peut être considéré comme un ensemble de règles (sémantique) représentant le comportement de ses éléments atomiques (syntaxe). Compte tenu certaines données (entrées) et d'une instance syntaxique d'un langage de programmation (programme), une interprétation est un processus permettant de renvoyer un résultat (sortie) en transformant les données par des manipulations du programme effectuées conformément aux règles du langage.

Un parallèle légitime peut être fait avec les langues naturelles. En effet, si avec une langue naturelle on construit des discours, interprétables par les interlocuteurs, avec les langages de programmation on produit des programmes, des directives pour la machine. Pour comprendre ces directives, il faut disposer d'un programme qui interprète le programme à la machine. Ce dernier doit être clair et sans ambiguïté, car l'interprétation du programme nécessite une représentation de la syntaxe et la sémantique dans laquelle ce programme est écrit.

Dans ce manuscrit, nous traitons de différents types de sémantique des langages de programmation et de leur représentation. Plus précisément, nous proposons un moyen efficace de représenter la sémantique des langages de programmation d'une manière qui soit concise et fidèle aux spécifications qui représentent leur comportement. Ces langages peuvent être de nature différente, tendant à être définis, de manière informelle, en langage naturel ou pseudo-algorithmique, ou, de manière formelle, en notation mathématique. Dans cette thèse, nous montrerons que, grâce à un petit métalangage appelé Skel, nous sommes capables de capturer la sémantique de ces deux types de spécifications. De ces définitions formelles, on extrait ensuite du code pour pouvoir produire de manière modulaire un interpréteur sur lequel on peut exécuter des programmes.

Définition de Langages Dans la Sémantique Squelettique.

Pour représenter formellement la définition d'un langage de programmation, il est généralement nécessaire de définir la syntaxe et la sémantique des constructions du langage.

Par exemple, si nous devions considérer un langage de programmation qui ne fait que des additions d'entiers, nous pourrions définir sa syntaxe de la manière suivante.

$$\begin{aligned} \textsc{Value}\ v &::= n \\ \textsc{Expr}\ e &::= v \mid e_1 + e_2 \end{aligned}$$

Avec cette représentation, on dit que les valeurs du langage sont les entiers, qui dans ce cas n'ont pas de définition réelle, et qu'une expression du langage est soit une valeur, soit une somme entre deux expressions.

En ce qui concerne la sémantique des expressions dans ce langage, il existe les différents formalismes qui peuvent être utilisés. Dans cette thèse, nous nous limiterons à sémantique opérationnelle, un cadre logique qui sert à décrire comment une construction de la langue est exécutée. Dans ce qui suit, nous présentons la sémantique du petit exemple de syntaxe défini ci-dessus.

$$\textsc{Valeur}\ \frac{}{n \Downarrow_e n} \qquad \textsc{Addition}\ \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 + n_2}$$

La première règle, `Valeur`, renvoie un nombre entier dans le cas où l'expression est simplement une valeur de la langue. La deuxième règle, `Addition`, évalue l'addition. Dans le cas d'une somme, il évalue le premier terme de l'addition, le second, et renvoie la somme des deux entiers. Comme la définition d'une expression est récursive, cette règle simple évalue également les sommes qui comportent d'autres additions entre des entiers en tant que sous-expressions. Dans la thèse, nous écrirons la plupart de la sémantique en utilisant la sémantique squelettique [8].

Pour donner immédiatement un aperçu de la puissance de la sémantique squelettique et du Skel, il faut considérer la conclusion de la règle, $n_1 + n_2$. Comme les entiers, l'addition n'a pas non plus de définition réelle, autre que celle que nous pourrions naturellement lui donner. Le langage Skel, est un petit langage fonctionnel défini

pour l'écriture de la sémantique. Exactement comme la définition formelle de ce langage avec l'addition, nous pouvons définir ce langage en Skel, avec le même niveau d'abstraction. Ce langage décrit la règle, et non le programme qui la représente.

```
type int
type value = | Int int
type expression = | Val value | Plus (expression, expression)

val plus : expression -> expression -> int

val eval (e:expression) : nat =
    branch let Val (Int n) = e in n
    or     let Add (e_1, e_2) = e in
           let n_1 = eval e_1 in
           let n_2 = eval e_2 in
           plus n_1 n_2
    end
```

En fait, en regardant le code, nous pouvons voir que nous définissons trois types, un pour les entiers, un pour les valeurs et un pour les expressions, mais contrairement aux deux derniers, les entiers, `type int`, n'ont pas de définition. Il en va de même pour l'addition `val plus`. Cette opération ne définit pas l'algorithme de la somme, elle indique simplement qu'il existe un terme qui prend deux entiers et renvoie un entier. Quant à la fonction `eval`, dans le cas où l'expression est une valeur, elle renvoie l'entier qu'elle encapsule, sinon, pour la somme, elle évalue la première expression, la seconde, et enfin renvoie la somme des deux entiers.

En général, pour des langages aussi petits, il est indifférent d'écrire la sémantique d'un langage de programmation sur papier ou sur une machine. Le problème se pose lorsque les langues sont de grande taille. En fait, certifier une grande sémantique sur le papier est presque impossible, ne serait-ce qu'en raison de la quantité de notation qui doit être définie afin d'avoir des formalisations compactes. Il ne faut pas non plus oublier que la démonstration par induction sur des définitions de langages aussi larges sur papier est compliquée. Au cours des dernières décennies, différentes approches ont été portées à l'état de l'art dans le but de pouvoir écrire la sémantique sur une machine, et de pouvoir utiliser le support de la machine pour certifier les inter-

prètes résultants. Il existe deux philosophies : l'une impliquant l'utilisation d'assistants de démonstration tels que Coq [70], HOL [29], Isabelle [30], Twelf [71, 27], et l'autre exploitant les logiciels et les environnements définis pour écrire la sémantique, tels que K [63], Ott [66], Lem [55], PLT [36]. La langue Skel est placée dans la deuxième catégorie. Contrairement aux concurrents de la deuxième catégorie, avec Skel on ne peut pas exécuter directement des programmes, mais on peut générer divers artefacts grâce à l'ensemble des outils fournis par `necro` [53].

Nous aborderons les styles et outils sémantiques dans la première partie de cette thèse.

Dans les parties 2 et 3, nous montrerons comment exploiter ce langage pour écrire des sémantiques pour des langages au comportement complexe.

Contributions

Les contributions de la thèse sont présentées dans les deuxième et troisième parties.

Dans la deuxième partie, les contributions sont de décrire une approche systématique pour écrire la sémantique d'une manière concise qui est fidèle à sa spécification, facilement maintenable, et à partir de laquelle un interprète peut être généré pour exécuter des programmes. Les langages que nous allons considérer ont des effets secondaires, tels que la modification de la mémoire, la présence d'exceptions, et suspensions du code. Dans cette partie de la thèse, nous présenterons d'abord une série de langages jouets et leur formalisation en Skel. Ces langages seront une extension de l'autre, et leur formalisation restera structurellement la même, déléguant la propagation des effets secondaires aux opérateurs monadiques. Nous présenterons plus tard la mécanisation d'ECMAScript. Nous utiliserons le même modus operandi pour produire une description formelle structurellement solide de cette spécification. La contribution dans ce cas est une formalisation concise, maintenable et lisible de la sémantique de l'ECMAScript; la génération d'un interpréteur fonctionnel et l'exécution de petits programmes.

Dans la troisième partie de la thèse, la contribution consiste à raffiner une sémantique déjà existante représentant le comportement d'un système distribué particulier. En fait, le contexte de ce système est celui des applications web qui incluent des interactions avec des dispositifs connectés au réseau. Nous appelons généralement ces systèmes *Internet des Objets* (IoT). Une autre contribution de cette partie est

la production d'un interprète pour cette sémantique, montrant que ce modèle formel non-déterministe pose des problèmes lors de son exécution, car il est complexe. Une dernière contribution de cette partie est la définition d'un ordonnanceur pour ce modèle qui limite le non-déterminisme, rendant le modèle exécutable. Pour soutenir la correction de l'ordonnanceur, nous fournirons un théorème d'équivalence entre la sémantique non-déterministe et celle de l'ordonnanceur.

Les contributions de la deuxième partie de la thèse sont présentées dans deux articles publiés et un en cours de révision [34, 33, 35].

Les contributions de la troisième partie de la thèse sont actuellement en cours de rédaction sous forme d'article.

Plan du Manuscrit

La thèse se compose de trois parties.

Le premier présentera, dans le chapitre 1, les techniques standard pour représenter formellement le comportement des constructions linguistiques d'un langage de programmation, et dans le chapitre 2, les logiciels créés pour avoir une représentation formelle de ces langages sur une machine. En détail, ce dernier chapitre présentera la sémantique des squelettes et le langage Skel, un petit langage fonctionnel permettant d'écrire des spécifications de langage.

La deuxième partie de la thèse traitera de la manière de capturer les effets secondaires des spécifications du langage. Par exemple, les langages peuvent avoir un état modifié, gérer les exceptions, suspendre l'exécution de certains segments de code. Ces effets sont généralement peu pratiques à gérer, car ils doivent être représentés explicitement dans un langage purement fonctionnel.

Dans le chapitre 3, nous verrons comment définir de manière modulaire la sémantique d'un petit langage de programmation jouet, que nous étendrons progressivement en introduisant de nouvelles constructions de langage et des effets secondaires. Nous montrerons que grâce à l'utilisation des monades, nous serons en mesure d'écrire des représentations extrêmement modulaires, solides et facilement extensibles en Skel. La propagation des effets secondaires sera gérée, implicitement, par les opérateurs monadiques.

Dans le chapitre 4, nous utiliserons les techniques d'écriture sémantique définies dans le chapitre précédent pour représenter la sémantique d'un langage réel, JavaScript.

Contrairement au chapitre précédent, où les spécifications du langage étaient décrites par des règles d'inférence logique, dans ce chapitre nous nous baserons sur une spécification textuelle. Contrairement à d'autres langages, comme Python, la spécification de JavaScript est extrêmement précise. Nous montrerons que notre technique d'écriture parvient à capturer, de manière fidèle, concise et maintenable, le comportement du langage. A partir de la représentation formelle, nous allons générer un interpréteur de noyau JavaScript et montrer l'exécution d'un petit programme. Pouvoir représenter l'ensemble de la spécification est utopique, en fait notre principale contribution est de montrer que le langage est suffisamment expressif pour rendre cette tâche historiquement difficile relativement simple.

La troisième partie de ce document présentera, au chapitre 5, une sémantique représentant le comportement d'un système distribué. Ce modèle, cherche à exprimer l'évolution de ces systèmes dans le contexte de l'Internet des objets. Nous allons montrer la sémantique, d'abord à travers les règles d'inférence et ensuite en Skel. Nous discuterons de la lourdeur de l'exécution de ce modèle, qui est non-déterministe par nature, et présenterons une solution, un ordonnanceur qui tentera de limiter le non-déterminisme inhérent à ce modèle. Nous montrerons que la sémantique du l'ordonnanceur est équivalente à la sémantique non-déterministe, ce qui rend possible l'exécution de programmes exécutés sur ce modèle, quelle que soit la complexité du programme.

TABLE OF CONTENTS

LIST OF FIGURES

A papá, mamma ed Amin,
un enorme piccola famiglia.

Introduction

« Try not. Do. Or do not. There is no try. »
- Yoda's unambiguous advice

Every programming language can be seen as a set of rules (*semantics*) representing the behavior of its atomic elements (*syntax*). Given some data (*inputs*) and a syntactical instance of a programming language (*program*), an *interpretation* is a process of returning a result (*output*) by transforming the data via program manipulations done according to the language rules.

The latter description is quite natural, as it is essentially similar to what happens with natural language human interpretation, such as English, Italian, Chinese, etc. In fact, given hypotheses, a requirement of a *natural program* (*context*), the language (defined by its *grammatical* rules), the meaning of a speech, or a thesis, is derived depending on each one's subjective understanding. Everyone has a slightly different interpretation of it, as subjectivity is inevitable for human beings. Indeed, while interpreting a natural program, everyone introduces some biases, such as misunderstandings from a lack of knowledge of the language, cultural background, personal judgment about the speaker/writer, and so on. Moreover, sentences like *"I saw someone on the hill with a telescope"* are syntactically and semantically/grammatically correct. Still, a certain ambiguity is intrinsically related to the sentence and the language itself. Indeed, one might ask who actually carried the telescope, the subject or the person on the hill?
To sum up, the difference between programming and natural languages is the ambiguity that the latter intrinsically has. Something that a programming language interpretation cannot afford. In the case of programming languages, if a program is syntactically and semantically correct, we should be able to say that we can expect a result of a specific type. Even non-deterministic programming languages, unclear in their behavior, must have a precise formal definition. For example, the parallel construct is defined in terms of non-deterministic interleavings between two programs or instructions.

Over the years, many works tried to formalize the behavior of programming languages in different styles and with various technologies, trying to make the definitions precise to enable studies spreading from proving the formal properties of a language to security or analysis of programs. The need to write a mathematical formalization pushed the definition of different semantic styles in which it is possible to capture the language features and technologies in which a language can be formally encoded. These technologies go from the classic paper to the use of the machine, all with some advantages and disadvantages. Still, the trend is to simplify both the complexity and

the learning curve to access one of these methods.

Writing a formalization on paper works great on small calculi, as scientists can directly provide a formal meaning to the syntactical elements of a language by writing the behavior in a semantic style. Still, the complexity of writing full-scale programming languages on paper is challenging. These definitions become easily massive, non-manageable, or writable, and coherence can be challenging to maintain. Then informal proofs, the ones made by hand, can be pretty unreliable as the language formalization grows. Assuming that the work is safe and sound after tremendous efforts, it is difficult to derive an implementation from it, making it hard to test concretely. We would like to recall that this scientific area aims to find formalisms and technologies to reduce deficiencies in programming language specifications and implementations, goal that we cannot be sure to be hit by simply unit-test interpreters or compilers.

The machines' support has been an important breakthrough for semanticists. For example, via machine, one can automatically check the coherence of the formalization while writing it. In the last three decades, different tools and techniques machine-checked spread out. We can factorize them in two different macro-approaches: Semantics via *proof assistants*, such as Coq [70], HOL [29], Isabelle [30], or Twelf [71, 27], and Semantics via *specification languages*, such as Ott [66], Lem [55], K [63], or PLT [36].

The use of proof assistants allows to formally verify that programming languages uphold certain language-specific properties. In general, these frameworks are designed to reason about formal properties of mathematical objects. Excellent work has been done on the proof assistants; CompCert [18, 5, 4] for C, CakeML [12, 48, 69] for StandardML [68, 44], and JSCert project [7, 6] for JavaScript are some strong examples. Nevertheless, these technologies, despite their great power, work with their own logic, making the portability of a work, from a proof assistant to another almost impossible. The sources are cluttered with syntactic noise and generally, one needs to know at best the tool. Moreover, these are complex tools, making hard to be fluent in more than one tool, as the learning curve is steep. The result is a lack of exchange between different communities of proof assistants, as these technologies have created different schools of thought. This impacts collaboration between scientists, not being able to port these accomplishments into different platforms, even automatically. Indeed, these languages are complex, making mechanisms of executable code extraction hard to design and to be proven correct. For example, works such as the JSCert project put trust on the Coq

extraction method to OCaml [19], without having any formal guarantee of semantics mismatches.

This thesis is set in the context of semantics via specification languages. These languages, generally meta-languages, are built on theoretical frameworks, each with its peculiarities, to provide an intuitive way to implement the semantics of a programming language. Generally tiny languages, these works offer tools to generate artifacts to execute and formally study a formalization.

In the thesis, we will focus on the Skeletal Semantics framework. After providing a bit of background about semantic styles and how to use them to encode the behavior of a variant of a Vanilla While language in Chapter 1, in Chapter 2 we present the Skeletal Semantics and the tools for producing artifacts. These two chapters will offer a solid background to the thesis.

Then, in Chapter 3, we will present an approach for writing semantics in Skel, the language for writing Skeletal Semantics. This work is currently under major revision by the authors[35]. We claim our techniques produce concise, readable, and maintainable semantics. This novel approach to specification languages allows exploiting continuations in our meta-language to design carefully chosen monad and hide side-effects in the programming language formalizations. Formalizations are structurally solid, and extending them means encoding the evaluation behavior of the language constructs and updating the monads. The core of the language stays immutated.

Chapter 4 presents the work published in the proceedings of the *24th International Symposium on Principles and Practice of Declarative Programming* and of the *32nd Journées Francophones des Langages Applicatifs* [34, 33]. We apply the technique presented in Chapter 3 by providing a formal semantics of ECMAScript and showing how to use our monadic approach to build a solid and extensible formalization of the JavaScript specification.

Finally, in Chapter 5, we present a model that aims to embody the behavior of a distributed system for IoT applications. This orchestrating semantics, tiny in size, models complex interactions between tiers, and writing it in Skel is still a challenge. Nevertheless, we propose the semantics and a naïve implementation. Then, to constrain the model's non-determinism, we produce a scheduling policy easily implementable in Skel, proven correct and complete.

Part I

Background

Chapter 1

Semantic Styles

Contents

« A thing named, misnamed, unnamed, or renamed is still itself. »
- Mokokoma Mokhonoana

Introduction

We define a programming language by its *syntax*, which is related to the grammatical structure of programs, and *semantics*, which is more concerned with expressing the meaning of a program. As the thesis presents an efficient and concise approach for capturing semantics, this chapter provides some background on general techniques that have historically and recently been used for defining the behavior of programming language constructs. Generally speaking, the goal of a *formal semantics* is to define rigorously/mathematically the meaning of the programs written in the considered programming language. The task is quite challenging, especially considering complex programming languages. Nevertheless, the implications are interesting, as formalizations can be declined in multiple applications of interest. For example, a formalization helps reveal ambiguities or inconsistencies in a programming language specification. It can also be a valid support for writing ground-truth implementations of interpreters.

Syntax of the `While` Language

$$
\begin{array}{rcl}
\textsc{Num}\ n & \in & \mathbb{Z}, \text{ extensionally } \{\ldots, -1, 0, 1, \ldots\} \\
\textsc{Bool}\ b & \in & \{T, F\}, \text{ respectively } \texttt{true} \text{ and } \texttt{false} \\
\textsc{Value}\ v & ::= & n \mid b \\
\textsc{Var}\ x & \in & \Sigma^+, \text{ where } \Sigma \text{ is an alphabet of our choice. i.e. } \{a, b, \ldots, z\} \\
\textsc{Store}\ \sigma & : & \textsc{Var} \to \textsc{Value} \\
\textsc{Expr}\ e & ::= & v \mid x \mid e_1 \diamond e_2 \mid e_1 \circ e_2 \mid e_1 \odot e_2 \mid !e \\
\textsc{Cmd}\ c & ::= & \texttt{skip} \mid x := e \mid c_1; c_2 \mid \texttt{if}(e)\{c_1\}\ \texttt{else}\ \{c_2\} \\
 & & \texttt{while}(e)\{c\}
\end{array}
$$

Figure 1.1: Definition of the `While` language grammar.

Moreover, often tools are built around these representations to perform, for example, analysis of programs and formal verification.

In this chapter, we introduce different ways of describing formally programming languages semantics and the notation used throughout all thesis. In detail, Section 1.1 presents a simple `While` language. Then, we present first the two operational semantics (Section 1.2), in Sections 1.2.1, 1.2.2, respectively, the Plotkin's *Structural Operational Semantics* (SOS), the *Natural Semantics* (NS), moving then to the *Abstract Machines* (AM) in Section 1.3, and the non-classical *Pretty-Big-Step Semantics* (PBS) in Section 1.4. For each of the semantic styles, we embed the `While` language variant presenting semantic rules for the expressions and statements, highlighting the limitations of each formal framework. This chapter aims to show textbook approaches to semantics as we base the design of the languages we present in the thesis on these semantic styles.

1.1 A While Language

The `While` language is a simple imperative language. In Figure 1.1 we define its syntax via a simplified *Backus–Naur form* [3] (BNF).

A VALUE can be either an integer or a booleans. We define the store as a partial map from variables to values. If a variable x is not bound in σ, the operation $\sigma(x)$ fails.

Otherwise, it returns the bound value. Given a variable x, we define $\sigma + x \to v$ as the environment extension or update, binding x to a value v.

Expressions can be values, variables, binary arithmetic operations (denoted as $\diamond$), logical binary operations (denoted as $\circ$), or the ! operator. We define $\diamond \in \{+, -, *, /\}$ and $\circ \in \{\geq, >, \leq, <, =, \wedge, \vee\}$. We call the concrete logical binary operations on integers and booleans, respectively $\circ_{\mathbb{Z}}$ and $\circ_{\mathbb{B}}$, to denote comparisons between integers or between booleans. Similarly, we define $\diamond_{\mathbb{Z}}$ and $\odot_{\mathbb{B}}$ as the concrete operations for computing arithmetic and unary expressions, respectively, between integers and on booleans. Notice that we partially define these operators, as we only allow evaluations producing values of the type denoted by the subscript. By design, a legal program can be an expression that reduces to, for example, $T+4$. We can freely decide on a strategy to evaluate the binary expressions. An approach can be one that does not allow such a thing to happen. We can define a partial function that only produces a result when the two subexpressions reduce to the same type. $T + 4$ fails in this case, as we chose to define operators partially. Otherwise, one can enrich the language, making the evaluation return either the expected result or an exception. This approach can be interesting for capturing non-allowed behaviors, such as division by 0. Finally, one can allow these mixed expressions to happen. This approach is more related to *dynamically typed* languages. A naïve strategy can be to evaluate the two subexpressions and give the result according to the type of the first value argument. This evaluation strategy must treat *unsafe coercion*s—implicit operations that change the values' type-.

For simplicity, we present a partial semantics, meaning that if no rule reduces a language constructor, the evaluation becomes stuck.

Once we reduce the two subexpressions for arithmetic operators, we can check both values to ensure type concordance. The result of an arithmetic expression is a numeric value. The result of an arithmetic expression is a numeric value.

We define the evaluation only with type concordance for the comparative logical expressions, concretely represented by $\circ_{\mathbb{Z}}$ and $\circ_{\mathbb{B}}$, while for the pure logical expressions, we allow only boolean values to be elements of the expression, concretely $\odot_{\mathbb{B}}$. We permit boolean comparison by saying that the relation $(T > F) = T$, and $(T \neq F) = T$. For what concerns the ! operator, if we negate a boolean, the function returns its complement. We proceed with the partial approach, allowing only the negation of a boolean.

To gently introduce evaluation descriptions, in Figure 1.2, we present a simple partial function ϕ that evaluates the expressions of the `While` language. Further encodings

$\phi : \mathbf{Expr} \times \mathbf{Store} \to \mathbf{Value}$

$$\phi(e,\sigma) = \begin{cases} \sigma(x), & \text{if } e = x, \text{ a variable} \\ v, & \text{if } e = v, \text{ a value.} \\ n_1 \diamond_{\mathbb{Z}} n_2, & \text{if } e = (e_1 \diamond e_2) \text{ and } \phi(e_1,\sigma) = n_1 \text{ and } \phi(e_2,\sigma) = n_2. \\ n_1 \circ_{\mathbb{Z}} n_2, & \text{if } e = (e_1 \circ e_2) \text{ and } \phi(e_1,\sigma) = n_1 \text{ and } \phi(e_2,\sigma) = n_2. \\ b_1 \circ_{\mathbb{B}} b_2, & \text{if } e = (e_1 \circ e_2) \text{ and } \phi(e_1,\sigma) = b_1 \text{ and } \phi(e_2,\sigma) = b_2. \\ b_1 \odot_{\mathbb{B}} b_2, & \text{if } e = (e_1 \odot e_2) \text{ and } \phi(e_1,\sigma) = b_1 \text{ and } \phi(e_2,\sigma) = b_2. \\ !b, & \text{if } e = !e' \text{ and } \phi(e',\sigma) = b. \end{cases}$$

Figure 1.2: Expression evaluation function ϕ

```
n := 3;b := 3;r := 1;
while(n > 0){
  r := r * b;
  n := n - 1
}
```

Figure 1.3: Example

of the expression's behavior will match the one described by this function.

The CMD syntactic production describes the commands of the language. These can be a `skip`, a variable assignment, a command sequencing, a conditional `if-else`, or a loop statement `while`.

In the listing in Figure 1.3, we show an example of a program. It computes in `r` the n-th power of a number b, 3^3 in the example.

1.2 Operational Semantics

An Operational Semantics of a programming language describes how programs execute. The approach consists in formally defining the meaning of each of the syntactic constructs of the language via logical relations. The abstraction helps avoid details related to the description of the machine that is supposed to execute a program. Indeed, concepts like the memory model, the machine architecture, and the implemen-

tation details of an interpreter are irrelevant, as the goal is to describe how to derive meaning from a language construct rather than a concrete implementation. To resume, operational semantics only describes what happens if a construct of the language is evaluated.

In general, for the operational semantics, a transition relation produces a *judgment* from one program configuration to another by applying an inference rule.

An *inference rule* has a set of *premises* on top of the line and the *conclusion* below the line. The conclusion is valid if all the premises are satisfied. Then, an inference rule without premises is called *axiom*, which is always true. We show them below.

$$\textsc{Rule}\quad \frac{premise \qquad premise \qquad premise}{conclusion} \qquad\qquad \textsc{Axiom}\quad \frac{}{conclusion}$$

A *judgment* is the result of a derivation built as a derivation tree. This structure represents the evaluation of an expression of the language. The root of this tree is the original expression, and each premise is a subtree of the derivation tree, justifying the evaluation of the expression. The conclusion is the resulting semantic judgement (evaluation) when the premises are satisfied. A conclusion is placed as the root of the derivation tree.

In the following two sections, we will present the *Structural Operational Semantics* and the *Natural Semantics* operationally, meaning that we will introduce them by formalizing the `While` language.

1.2.1 Structural Operational Semantics

The Structural Operational semantics goal is to describe computations in terms of their individual steps. Indeed, SOS is also known as Small-Step semantics. In 1981, Plotkin wrote the famous “Aarhus” notes [61], which first presented the *Structural* approach for defining the “operational”– the meaning of the program - semantics via logical statements rather than partial functions, as done by using the *denotational semantics* [73, 43]. The intention behind proposing this formal approach was to provide logical-relational meaning to the constructs of a programming language rather than a functional one. Indeed, the operational behavior of a programming language then is considered to be an abstraction, rather than a description, that could be used for proof

Store $\times$ **Expr** $\downarrow_e$ **Store** $\times$ **Expr** $-$ **Store** $\times$ **Expr** $\downarrow_e$ **Value**

$$\textsc{Variable}\ \frac{\sigma(x) = v}{(\sigma, x) \downarrow_e v} \qquad \textsc{AritHExprL}\ \frac{(\sigma, e_1) \downarrow_e (\sigma, e_1')}{(\sigma, e_1 \diamond e_2) \downarrow_e (\sigma, e_1' \diamond e_2)} \qquad \textsc{AritHExprR}\ \frac{(\sigma, e_2) \downarrow_e (\sigma, e_2')}{(\sigma, n_1 \diamond e_2) \downarrow_e (\sigma, n_1 \diamond e_2')}$$

$$\textsc{AritHExpr}\ \frac{n = n_1 \diamond_{\mathbb{Z}} n_2}{(\sigma, n_1 \diamond n_2) \downarrow_e n} \qquad \textsc{CompExprL}\ \frac{(\sigma, e_1) \downarrow_e (\sigma, e_1')}{(\sigma, e_1 \circ e_2) \downarrow_e (\sigma, e_1' \circ e_2)} \qquad \textsc{CompExprR}\ \frac{(\sigma, e_2) \downarrow_e (\sigma, e_2')}{(\sigma, v_1 \circ e_2) \downarrow_e (\sigma, v_1 \circ e_2')}$$

$$\textsc{CompExprN}\ \frac{n = n_1 \circ_{\mathbb{Z}} n_2}{(\sigma, n_1 \circ n_2) \downarrow_e n} \qquad \textsc{CompExprB}\ \frac{b = b_1 \circ_{\mathbb{B}} b_2}{(\sigma, b_1 \circ b_2) \downarrow_e b} \qquad \textsc{logExprL}\ \frac{(\sigma, e_1) \downarrow_e (\sigma, e_1')}{(\sigma, e_1 \odot e_2) \downarrow_e (\sigma, e_1' \odot e_2)}$$

$$\textsc{logExprR}\ \frac{(\sigma, e_2) \downarrow_e (\sigma, e_2')}{(\sigma, b_1 \odot e_2) \downarrow_e (\sigma, b_1 \odot e_2')} \qquad \textsc{logExpr}\ \frac{b = b_1 \odot_{\mathbb{B}} b_2}{(\sigma, b_1 \odot b_2) \downarrow_e b} \qquad \textsc{NegE}\ \frac{(\sigma, e) \downarrow (\sigma, e')}{(\sigma, !e) \downarrow_e (\sigma, !e')} \qquad \textsc{NegB}\ \frac{}{(\sigma, !b) \downarrow_e \neg b}$$

Figure 1.4: Small-Step expression evaluation. This description matches the behavior defined in Figure 1.2.

of correctness, safety, or security.

Usually, a language construct's semantics can be defined in terms of the behavior of its parts, driving the evaluation by its syntactic structure. Then, the SOS program's evaluation can be seen as a sequence of transition relations that reduce the program to its final result. Indeed, each syntactic construct of a language has a set of inference rules that define its behavior–set of valid *judgments*- and each valid judgment, in this example, modifies and reduces the shape of the program. The program finishes evaluating when it is not possible to perform another transition. Thus, small-step semantics are ideal for watching the program execute step by step.

We define two different transition relations, one for evaluating expressions and the other for commands.

In the case of expressions, $\downarrow_e$ is a semantic judgment expressed as a relation between expression *configurations* $(\sigma, e) \downarrow_e (\sigma, e')$, or a relation between an expression configuration and a value $(\sigma, e) \downarrow_e v$. A configuration is a pair $(\textsc{Store}, \textsc{Expr})$. In Figure 1.4, we present a small-step operational semantics for evaluating expressions, according to the equational definition depicted in Figure 1.2.

Store $\times$ **CMD** $\downarrow_c$ **Store** $\times$ **CMD** $-$ **Store** $\times$ **Expr** $\downarrow_c$ **Store**

$$\textsc{Skip}\ \frac{}{(\sigma, \texttt{skip}) \downarrow_c \sigma} \qquad \textsc{Assign}\ \frac{(\sigma, e) \downarrow_e (\sigma, e')}{(\sigma, x := e) \downarrow_c (\sigma, x := e')} \qquad \textsc{AssignV}\ \frac{}{(\sigma, x := v) \downarrow_c \sigma + (x \to v)}$$

$$\textsc{SeqL}\ \frac{(\sigma, c_1) \downarrow_c (\sigma', c_1')}{(\sigma, c_1; c_2) \downarrow_c (\sigma', c_1'; c_2)} \qquad \textsc{SeqR}\ \frac{(\sigma, c_1) \downarrow_c \sigma'}{(\sigma, c_1; c_2) \downarrow_c (\sigma', c_2)}$$

$$\textsc{If}\ \frac{(\sigma, e) \downarrow_e (\sigma', e')}{(\sigma, \texttt{if}(e)\{c_1\}\texttt{else}\{c_2\}) \downarrow_c (\sigma', \texttt{if}(e')\{c_1\}\texttt{else}\{c_2\})}$$

$$\textsc{IfT}\ \frac{}{(\sigma, \texttt{if}(T)\{c_1\}\texttt{else}\{c_2\}) \downarrow_c (\sigma, c_1)} \qquad \textsc{IfF}\ \frac{}{(\sigma, \texttt{if}(F)\{c_1\}\texttt{else}\{c_2\}) \downarrow_c (\sigma, c_2)}$$

$$\textsc{While}\ \frac{}{(\sigma, \texttt{while}(e)\{c\}) \downarrow_c (\sigma, \texttt{if}(e)\{c; \texttt{while}(e)\{c\}\}\texttt{else}\{\texttt{skip}\})}$$

Figure 1.5: Small-Step commands evaluation.

Regarding commands, we define the relation $\downarrow_c$ to be either a relation between command configurations $(\sigma, c) \downarrow_c (\sigma', c')$ or a relation $(\sigma, c) \downarrow_c \sigma'$. Similarly, a configuration is a couple $(\textsc{Store}, \textsc{Cmd})$. Regarding the store, we define two operations. The first is for getting a binding from the store: given a store σ and a variable x, $\sigma(x)$ returns a value. The second operation is the store extension/update: given a store σ, variable x, and a value v, $\sigma + (x \to v)$ returns the store σ updated with the binding $x \to v$. We present the definition of the `While` language behavior in Figure 1.5.

If we consider the `if-else` construct rules, we notice that for evaluating it, first, we have to evaluate the guard condition. As we know, it might take several steps. Once the condition is evaluated to a value, the rules IfT and IfF put the first or second branch into evaluation. No reductions are yet done on c_1 or c_2. For the `while` command evaluation, we notice that the While rule changes only the program's structure to an `if-else`. This judgment changes the structure of the initial syntactic construct without performing any reduction of the statement itself. It is one of the reasons that this semantics is called structural, as it reasons on the structure of the program, sometimes by reducing

it, and sometimes by transforming it to something equivalent.

In the following listing, we show concretely the use of a small-step semantics to reduce a program. We consider the program presented in Figure 1.3, initially having an empty store σ, denoted as $\emptyset$.

($\emptyset$, `n := 3; b := 2; r := 1; while(b > 0){ r := r * n; b := b - 1 }`)

Then, the following shows the program reductions.

$\downarrow_c$ ([n→3], `b := 2; r := 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→2], `r := 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→2,r→1], `while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→2,r→1], `if(b > 0){ r := r * n; b := b - 1; while(b > 0){...}}else{skip}`)

$\downarrow_c$ ([n→3,b→2,r→1], `if(2 > 0){ r := r * n; b := b - 1; while(b > 0){...}}else{skip}`)

$\downarrow_c$ ([n→3,b→2,r→1], `if(T){ r := r * n; b := b - 1; while(b > 0){...}}else{skip}`)

$\downarrow_c$ ([n→3,b→2,r→1], `r := r * n; b := b - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→2,r→1], `r := 1 * n; b := b - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→2,r→1], `r := 1 * 3; b := b - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→2,r→1], `r := 3; b := b - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→2,r→3], `b := b - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→2,r→3], `b := 2 - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→2,r→3], `b := 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→1,r→3], `while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→1,r→3], `if(b > 0){ r := r * n; b := b - 1; while(b > 0){...}}else{skip}`)

$\downarrow_c$ ([n→3,b→1,r→3], `if(1 > 0){ r := r * n; b := b - 1; while(b > 0){...}}else{skip}`)

$\downarrow_c$ ([n→3,b→1,r→3], `if(T){ r := r * n; b := b - 1; while(b > 0){...}}else{skip}`)

$\downarrow_c$ ([n→3,b→1,r→3], `r := r * n; b := b - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→1,r→3], `r := 3 * n; b := b - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→1,r→3], `r := 3 * 3; b := b - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→1,r→3], `r := 9; b := b - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→1,r→9], `b := b - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→1,r→9], `b := 1 - 1; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→1,r→9], `b := 0; while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→0,r→9], `while(b > 0){ r := r * n; b := b - 1 }`)

$\downarrow_c$ ([n→3,b→0,r→9], `if(b > 0){ r := r * n; b := b - 1; while(b > 0){...}}else{skip}`)

$\downarrow_c$ ([n→3,b→0,r→9], `if(0 > 0){ r := r * n; b := b - 1; while(b > 0){...}}else{skip}`)

$\downarrow_c$ ([n→3,b→0,r→9], `if(F){ r := r * n; b := b - 1; while(b > 0){...}}else{skip}`)

$\downarrow_c$ ([n→3,b→0,r→9], `skip`)

$\downarrow_c$ [n→3,b→0,r→9]

We can see that this program computes $r = n^b$. For the initial assignments of `n = 3`, `b = 2`, and `r = 1`, we get `r = 9`, as expected.

Notice that each small step performed on the program configuration is a derivation tree for SOS. The sequence of transitions is a sequence of derivation trees, and each derivation tree is a small-step judgment. Indeed, the number of derivation trees equals the number of transition relations applied. Then, each tree has as many subtrees as the premises of the rule applied by that transition relation. Intuitively, an infinite sequence of derivation steps on a program represents a diverging computation.

1.2.2 Big-Step Operational Semantics

Different from small-step semantics, big-step, originally called *natural semantics* [32], is more concerned with the relationship between an execution's initial and final configuration. Indeed, here the description of the semantics focuses on describing the overall execution of a program rather than tracing each step. Then a big-step judgment can successfully give a result, gets stuck as no rule applies, or never terminate. Notice that, as the natural semantics focuses more on representing the overall judgment in one single big step, there is no way to understand if a derivation is stuck or does not terminate. Usually, languages represented in big-step semantics result in a more concise rule-set.

For what concerns expressions, differently from what we have presented in Section 1.2.1, we now evaluate expressions in one step. So, first, we re-define the transition relation $\downarrow_e$, as $\Downarrow_e$. The new relation, relates expression configurations $(\textsc{Store}, \textsc{Expr})$ to values. The VARIABLE rule does not change much, and we present it below.

$$\textsc{Variables}\ \frac{\sigma(x) = v}{\sigma, x \Downarrow_e v}$$

Let us consider comparison expressions, which are the expressions with more related rules. We can see that the number of rules reduces dramatically by bunching

Store × CMD $\Downarrow_c$ Store

$$\textsc{Skip}\ \frac{}{(\sigma, \texttt{skip}) \Downarrow_c \sigma} \qquad \textsc{Assign}\ \frac{(\sigma, e) \Downarrow_e v}{(\sigma, x := e) \Downarrow_c \sigma + (x \to v)} \qquad \textsc{Seq}\ \frac{(\sigma, c_1) \Downarrow_c \sigma'' \quad (\sigma'', c_2) \Downarrow_c \sigma'}{(\sigma, c_1; c_2) \Downarrow_c \sigma'}$$

$$\textsc{IfT}\ \frac{(\sigma, e) \Downarrow_e T \quad (\sigma, c_1) \Downarrow_c \sigma'}{(\sigma, \texttt{if}(e)\{c_1\}\texttt{else}\{c_2\}) \Downarrow_c \sigma'} \qquad \textsc{IfF}\ \frac{(\sigma, e) \Downarrow_e F \quad (\sigma, c_2) \Downarrow_c \sigma'}{(\sigma, \texttt{if}(e)\{c_1\}\texttt{else}\{c_2\}) \Downarrow_c \sigma'}$$

$$\textsc{WhileT}\ \frac{(\sigma, e) \Downarrow_e T \qquad (\sigma, c) \Downarrow_c \sigma'' \quad (\sigma'', \texttt{while}(e)\{c\}) \Downarrow_c \sigma'}{(\sigma, \texttt{while}(e)\{c\}) \Downarrow_c \sigma'} \qquad \textsc{WhileF}\ \frac{(\sigma, e) \Downarrow_e b \quad b = F}{(\sigma, \texttt{while}e\ \{c\}) \Downarrow_c \sigma}$$

Figure 1.6: Commands Evaluation in Big-Step

the premises of the rules together.

$$\textsc{CompExprB}\ \frac{(\sigma, e_1) \Downarrow_e b_1 \quad (\sigma, e_2) \Downarrow_e b_2 \quad b = b_1 \circ_{\mathbb{B}} b_2}{(\sigma, e_1 \circ e_2) \Downarrow_e b} \qquad \textsc{CompExprN}\ \frac{(\sigma, e_1) \Downarrow_e n_1 \quad (\sigma, e_2) \Downarrow_e n_2 \quad n = n_1 \circ_{\mathbb{Z}} n_2}{(\sigma, e_1 \circ e_2) \Downarrow_e n}$$

Considering the new relation $\Downarrow_e$,we can immediately notice that a rule now is written in terms of evaluating its subterms, describing the overall result of the evaluation as a final judgment. In Figure 1.6, we present the big-step definition of the `While` language. The evaluation relation $\Downarrow_c$ is between a configuration (σ, c) and a store σ.

Then, evaluating the program depicted in Figure 1.3 results simply in the following.
(∅, `n := 3; b := 2; r := 1; while(e > 0){ r := r * n; b := b - 1 }`)
$\Downarrow_c$ [n→3,b→0,r→9]
Notice that, differently from the small-step semantics, in this case, the overall derivation is no more a sequence of relation applications. However, it is a unique derivation tree containing all the other judgements, the ones of its subtrees. We present a partial

$$\textsc{Seq}\ \frac{\textsc{Assign}\ \frac{(\emptyset, 3) \Downarrow_e 3}{(\emptyset, \texttt{n:=3}) \Downarrow_c \emptyset + n \to 3}\ \textsc{Seq}\ \frac{\textsc{Assign}\ \frac{([n \to 3], \texttt{b:=2}) \Downarrow_e 2}{(\emptyset, \texttt{b:=2}) \Downarrow_c [n \to 3] + e \to 2}\ \textsc{Seq}\ \frac{\ldots}{([n \to 3, b \to 2], \texttt{r := 1; ...}) \Downarrow_c [n \to 3, b \to 0, r \to 9]}}{([n \to 3],\ \texttt{b := 2; r := 1; ...}) \Downarrow_c [n \to 3, b \to 0, r \to 9]}}{(\emptyset,\ \texttt{n := 3; b := 2; r := 1; while(b > 0)\{ r := r * n; b := b - 1 \}}) \Downarrow_c [n \to 3, b \to 0, r \to 9]}$$

Figure 1.7: First level of the big-step derivation tree of the program. We put ... when we refer to the rest of the program.

derivation tree in Figure 1.7.

Moreover, looking back at the rules, different from the SOS, natural semantics seems more structured on where to place the premises in a rule. Indeed, one can see premises more like a sequence rather than a set.

The big-step semantics does not perform any syntactic transformation of terms. An example is the `while` command, which, differently from the SOS—term transformation into an `if`-, is "naturally" evaluated.

1.3 Abstract Machines

The correctness of programming language implementations is an important issue confronting language designers and implementors. Traditionally, such implementations are handcrafted first and only then proved correct. Unfortunately, keeping a strong relationship between a formal specification and its implementation may be challenging to show. Indeed, one must relate several details to its formal representation. Alternatively, a language implementation can be constructed from the semantic specification so that the resulting implementation guarantees correctness.

Abstract machines (AM) [38] can provide an intermediate representation of the language's implementation that remains loyal to its operational semantics [26]. As we have seen in Section 1.2, operational semantics can be defined in terms of inference rules. An equivalent abstract machine can be built as rewriting rules describing single-step operations on the state of a computation. Such specifications provide an intermediate level of representation for many practical implementations of programming languages. Abstract machines are traditionally constructed by hand, with correctness proofs that follow independently.

For building an AM for the `While` language, we first define three states/modes for the machine. The first two states set the machine to evaluate expressions or statements. We denote the two respectively as $\langle c, \pi, \sigma \rangle_{\mathsf{CMD}}$ or $\langle e, \pi, \sigma \rangle_{\mathsf{EXPR}}$, meaning that a command or an expression is evaluated in a state σ with a continuation stacked in π. The third mode signals the result to another computation frame waiting for a result. In the normal case, it communicates the value to the top pending computation frame, which is the top of the stack π. We denote two continuation modes. The first continuation mode $[\pi, \sigma, r]_c$ propagates a value to the top-continuation stacked in π in a state σ. The second continuation mode $[\pi, \sigma]_c$ returns the state—it is similar to return a unit result.

Expression Evaluation AM

$$\langle v, \pi, \sigma\rangle_{\text{EXPR}} \rightarrow [\pi, \sigma, v]_c \quad (1.1)$$
$$\langle x, \pi, \sigma\rangle_{\text{EXPR}} \rightarrow [\pi, \sigma, \sigma(x)]_c \quad (1.2)$$
$$\langle e_1 \diamond_C e_2, \pi, \sigma\rangle_{\text{EXPR}} \rightarrow \langle e_1, (\square \diamond_C e_2) :: \pi, \sigma\rangle_{\text{EXPR}} \quad (1.3)$$
$$[(\square \diamond_C e_2) :: \pi, \sigma, v_1]_c \rightarrow \langle e_2, (v_1 \diamond_C \square) :: \pi, \sigma\rangle_{\text{EXPR}} \quad (1.4)$$
$$[(b_1 \diamond_C \square) :: \pi, \sigma, b_2]_c \rightarrow [\pi, \sigma, b_1 \diamond_C b_2]_c \quad (1.5)$$
$$[(n_1 \diamond_C \square) :: \pi, \sigma, n_2]_c \rightarrow [\pi, \sigma, n_1 \diamond_C n_2]_c \quad (1.6)$$

Figure 1.8: Abstract Machine definition of expression evalauation.

An AM stops evaluating once there are no more things to do. Nothing to do means that π becomes empty. If no rule is applicable, then the machine gets stops.

For instance, an AM is defined by its rewriting relation $\rightarrow$, relating computations.

As we did for the expressions, in Section 1.2.2, Figure 1.8 presents the AM transition rules for variables, the comparative logical expressions, adding the rule that signals convergence to a value. For instance, if we consider the evaluation of a value v, rule 1.1, the transition switches the expression evaluation mode to the continuation one, signaling to the top of π, the value v. Similarly, evaluating a variable x, rule 1.2, means to switch to a continuation mode carrying, as a result, the value bound to x in σ, $\sigma(x)$. Then, for evaluating a comparative logical expression, when computing the first argument of $e_1 \diamond e_2$ (rule 1.3), we set e_1 to be evaluated and push $\square \diamond e_2$ onto the stack as the top pending evaluation. The expression with the hole is a frame stating that we are evaluating a comparative logical expression, and e_1 is set to evaluation. Once the evaluation of e_1 is complete, applying the frame $\square \diamond e_2$ to the value v_1 means switching back to evaluation mode for e_2 with a new stacked frame $v_1 \diamond \square$ (rule 1.4) on top of π. Notice that the frame $\square \diamond e_2$ is popped, and the top of the stack now stores the value v_1. Finally, applying $v_1 \diamond \square$ to a value v_2 computes $v_1 \diamond v_2$, depending on the type concordance of the values v_1 and v_2(rules 1.5 to 1.6), as defined in Figure 1.2. If the types of the values mismatch, the abstract machine gets stuck.

At this point, we can note a similarity to the small-step semantics, as the abstract machine evolves step by step. For example, rule 1.3, sets e_1 to evaluation. Then depending on the expression's complexity, we need to apply several different rules to converge to a result.

Commands Evalaution AM

$$\langle \texttt{skip}, \pi, \sigma \rangle_{\text{CMD}} \to [\pi, \sigma]_c$$
$$\langle x := e, \pi, \sigma \rangle_{\text{CMD}} \to \langle e, (x := \square) :: \pi, \sigma \rangle_{\text{EXPR}}$$
$$[(x := \square) :: \pi, \sigma, v]_c \to \langle \texttt{skip}, \pi, (\sigma + x \to v) \rangle_{\text{CMD}}$$
$$\langle c_1; c_2, \pi, \sigma \rangle_{\text{CMD}} \to \langle c_1, c_2 :: \pi, \sigma \rangle_{\text{CMD}}$$
$$\langle \texttt{if}(e)\{c_1\}\texttt{else}\{c_2\}, \pi, \sigma \rangle_{\text{CMD}} \to \langle e, (\texttt{if}(\square)\{c_1\}\texttt{else}\{c_2\}) :: \pi, \sigma \rangle_{\text{EXPR}}$$
$$[(\texttt{if}(\square)\{c_1\}\texttt{else}\{c_2\}) :: \pi, \sigma, T]_c \to \langle c_1, \pi, \sigma \rangle_{\text{CMD}}$$
$$[(\texttt{if}(\square)\{c_1\}\texttt{else}\{c_2\}) :: \pi, \sigma, F]_c \to \langle c_2, \pi, \sigma \rangle_{\text{CMD}}$$
$$\langle \texttt{while}(e)\{c\}, \pi, \sigma \rangle_{\text{CMD}} \to \langle \texttt{if}(e)\{c; \texttt{while}(e)\{c\}\}\texttt{else}\{\texttt{skip}\}, \pi, \sigma \rangle_{\text{CMD}}$$

Figure 1.9: Abstract Machine definition of commands evalauation.

If we consider an initial machine $\langle 1 < 2, \epsilon, [] \rangle_{\text{EXPR}}$,where ϵ is the empty stack, then we can derive the following sequence.

$$\begin{aligned}
&\langle 1 < 2, \epsilon, [] \rangle_{\text{EXPR}} \ \to \\
&\langle 1, (\square < 2) :: \epsilon, \sigma \rangle_{\text{EXPR}} \ \to \\
&[(\square < 2) :: \epsilon, \sigma, 1]_c \ \to \\
&\langle 2, (1 < \square) :: \epsilon, \sigma \rangle_{\text{EXPR}} \ \to \\
&[(1 < \square) :: \epsilon, \sigma, 2]_c \ \to \\
&[\epsilon, \sigma, 1 < 2]_c \quad = \quad [\epsilon, \sigma, T]_c
\end{aligned}$$

The expression results in `T`, as expected. Note that each subexpression is first set to evaluation, then concretely evaluated.

$$\begin{aligned}
&\langle \texttt{add}(1, 1), \epsilon \rangle_{\text{EXPR}} \ \to \ \langle 1, \texttt{add}(\square, 1) :: \epsilon \rangle_{\text{EXPR}} \ \to \\
&[\texttt{add}(\square, 1) :: \epsilon, 1]_c \ \to \ \langle 1, \texttt{add}(1, \square) :: \epsilon \rangle_{\text{EXPR}} \ \to \\
&[\texttt{add}(1, \square) :: \epsilon, 1]_c \ \to \ [\epsilon, 1]_c
\end{aligned}$$

Similarly, we can formalize commands semantics via AM. We present the abstract machine representing the semantics of the language in Figure 1.9.

1.4 Pretty-Big-Step Semantics

Our `While` language is dynamically typed, in the sense that there is no check that undefined operations such as $T + 4$ are absent, hence a program can become stuck. Moreover, we know that a big step evaluation of a program can be either infinite, in the case there is always a rule able to apply, stuck when there is none, or stop when it reaches a result. The problem for the Natural Semantics (NS) is that it is hard to know whether the evaluation is converging to a result or is stuck because something wrong happened—its original goal was to express the semantics of terminating programs. For example, following our previous language definitions, we have never considered handling the division by zero by allowing a failure in the evaluation.

If we consider the division by zero in NS, then we need to tweak the syntax and the semantics to carry the information about a failure to the top level and not simply get blocked somewhere in the middle of the program evaluation. For example, we can define an *option type* and say that a semantic judgment of an expression($\Downarrow_e$) either exists, denoted as `Some` v, where v is the result of evaluating an expression, or `None` when a failure, for instance, division by zero, happens. Now, consider the addition and the division rule considering the new big-step relation for expressions presented in Figure 1.10. We notice that, by only expressing the failure to the parent node in the derivation tree, we increase the by a factor of at least three the number of rules. The description of the addition consists of three rules: the first two, AddF1 and AddF2, state the failure of the evaluation of e_1 or e_2, and the third evaluates the addition (Add). For the division rules, things are pretty similar, having one rule more preventing division by 0. Notice the redundancy in the premises. The rules AddF1 and AddF2 are basically the same as DivF1, and DivF2.
Moreover, the Add, and the two Div and Div0 share most of the rule premises. To be precise, the number of $\Downarrow_e$ in the premises of the addition are 5, and 7 for the division. These redundancies can be really annoying while trying to prove by structural induction properties of the language. Indeed it leads to consider several times the same transition relation, proving more than once the same proof sub-tree. To sum up, the handling of the exceptions is explicit and redundant, making formalizations and proofs grow both in size and complexity even for one more piece of information carried around through derivations. So, can we factorize the big-step rules to avoid redundancies?

The answer is yes; we can do it by changing the formal definition of the program-

Store $\times$ Expr $\Downarrow_e$ Opt

$$\textsc{AddF1}\ \frac{\sigma, e_1 \Downarrow_e \texttt{None}}{\sigma, e_1 + e_2 \Downarrow_e \texttt{None}} \qquad \textsc{AddF2}\ \frac{\sigma, e_1 \Downarrow_e \texttt{Some}\ n_1 \quad \sigma, e_2 \Downarrow_e \texttt{None}}{\sigma, e_1 + e_2 \Downarrow_e \texttt{None}} \qquad \textsc{Add}\ \frac{\sigma, e_1 \Downarrow_e \texttt{Some}\ n_1 \quad \sigma, e_2 \Downarrow_e \texttt{Some}\ n_2 \quad n = n_1 + n_2}{\sigma, e_1 + e_2 \Downarrow_e \texttt{Some}\ n}$$

$$\textsc{DivF1}\ \frac{\sigma, e_1 \Downarrow_e \texttt{None}}{\sigma, e_1/e_2 \Downarrow_e \texttt{None}} \qquad \textsc{DivF2}\ \frac{\sigma, e_1 \Downarrow_e \texttt{Some}\ n_1 \quad \sigma, e_2 \Downarrow_e \texttt{None}}{\sigma, e_1/e_2 \Downarrow_e \texttt{None}} \qquad \textsc{Div0}\ \frac{\sigma, e_1 \Downarrow_e \texttt{Some}\ n_1 \quad \sigma, e_2 \Downarrow_e \texttt{Some}\ 0}{\sigma, e_1/e_2 \Downarrow_e \texttt{None}}$$

$$\textsc{Div}\ \frac{\sigma, e_1 \Downarrow_e \texttt{Some}\ n_1 \quad \sigma, e_2 \Downarrow_e \texttt{Some}\ n_2 \quad n_2 \neq 0 \quad n = n_1/n_2}{\sigma, e_1/e_2 \Downarrow_e \texttt{Some}\ n}$$

Figure 1.10: Expression Evaluation with Option Results in Big-Step

ming language. Arthur Charguéraud introduced the pretty-big-step semantics [15] to factorize the description of both effectful programming languages and managing to capture divergent programs while analyzing programs. This semantics preserves the big-step philosophy by keeping the overall reduction of a program happening in a single transition relation application. However, the big-step and the pretty-big-step derivation trees are quite different, as the PBS rules decouple syntactic constructs and effects behaviors. Let us try to write the addition and division rules, considering division by zero. The idea is to decouple the behavior of a syntactic construct from the actual effect that the evaluation of a subterm carries. We re-define only addition and division, but the approach must be extended to the rest of the arithmetic and logical expressions.

Expression Syntax in Pretty-Big-Step

$$\textsc{Opt}\ o ::= \texttt{None} \mid \texttt{Some}\ v$$

$$\textsc{Expr}\ e ::= o \mid x \mid e_1 + e_2 \mid o_1 +_1 e_2 \mid v_1 +_2 o_2 \mid e_1/e_2 \mid o_1\ /_1\ e_2 \mid v_1\ /_2\ o_2$$

Store $\times$ **Expr** $\Rightarrow_e$ **Opt**

$$\textsc{Add}\ \frac{\sigma, e_1 \Rightarrow_e o_1 \quad \sigma, o_1 +_1 e_2 \Rightarrow_e o}{\sigma, e_1 + e_2 \Rightarrow_e o} \qquad \textsc{AddF1}\ \frac{}{\sigma, \texttt{None} +_1 e_2 \Rightarrow_e \texttt{None}} \qquad \textsc{Add1}\ \frac{\sigma, e_2 \Rightarrow_e o_2 \quad \sigma, n_1 +_2 o_2 \Rightarrow_e o}{\sigma, (\texttt{Some}\ n_1) +_1 e_2 \Rightarrow_e o}$$

$$\textsc{AddF2}\ \frac{}{\sigma, n_1 +_2 \texttt{None} \Rightarrow_e \texttt{None}} \qquad \textsc{Add2}\ \frac{n = n_1 + n_2}{\sigma, n_1 +_2 (\texttt{Some}\ n_2) \Rightarrow_e \texttt{Some}\ n} \qquad \textsc{Div}\ \frac{\sigma, e_1 \Rightarrow_e o_1 \quad \sigma, o_1 /_1 e_2 \Rightarrow_e o}{\sigma, e_1 / e_2 \Rightarrow_e o}$$

$$\textsc{DivF1}\ \frac{}{\sigma, \texttt{None} /_1 e_2 \Rightarrow_e \texttt{None}} \qquad \textsc{Div1}\ \frac{\sigma, e_2 \Rightarrow_e o_2 \quad \sigma, n_1 /_2 o_2 \Rightarrow_e o}{\sigma, (\texttt{Some}\ n_1) /_1 e_2 \Rightarrow_e o} \qquad \textsc{DivF2}\ \frac{}{\sigma, n_1 /_2 \texttt{None} \Rightarrow_e \texttt{None}}$$

$$\textsc{Div0}\ \frac{}{\sigma, n_1 /_2 (\texttt{Some}\ 0) \Rightarrow_e \texttt{None}} \qquad \textsc{Div2}\ \frac{n_2 \neq 0 \qquad n = n_1 / n_2}{\sigma, n_1 /_2 (\texttt{Some}\ n_2) \Rightarrow_e \texttt{Some}\ n}$$

Figure 1.11: Expression Evaluation in Pretty-Big-Step

The presented syntax is a reformulation of the previous. This reformulation is a decomposition that aims to split a few complex and redundant rules into a more significant number of atomic rules. We show it in Figure 1.11. The pretty-big-step transition relation for expressions is $\Rightarrow_e$, and it relates expression configurations (σ, e) to option values o. To begin, we can say that now, for both the set of rules for addition and division, we have only 4 $\Rightarrow_e$ transitions in the premises. All these premises produce different proof sub-trees. Thus, this formalization of the expressions is no more redundant.

Consider the PBS' addition rules. The ADD rule first evaluates expression e_1, which results in an option value o_1, and then evaluates $o_1 +_1 e_2$. We can say that the rule formulation does not state any failure of the evaluation, while it simply delegates the check to another rule. Moreover, notice that the overall evaluation happens in a single derivation tree that grows in depth. Then, for evaluating $o_1 +_1 e_2$, one can apply the rule ADDF1 or ADD1 depending on whether the first expression failed or not. Notice that the syntactic operators $+_1$ and $+_2$ define intermediate evaluation stages of the

Expression Syntax in Pretty-Big-Step

$$
\begin{array}{rcl}
\textsc{Res}\ r & ::= & \texttt{Exc}\ \sigma \mid \sigma \\
\textsc{Cmd}\ c & ::= & \texttt{skip} \mid x := e \mid x :=_1 o \mid c_1; c_2 \mid r;_1\ c_2 \mid \texttt{if}(e)\{c_1\}\ \texttt{else}\ \{c_2\} \\
& & \mid \texttt{if1}(o)\{c_1\}\ \texttt{else}\ \{c_2\} \mid \texttt{while}(e)\{c\}
\end{array}
$$

Figure 1.12: Syntax definition of the language in Pretty-Big-Step style.

addition, respectively stating if either the first or the second expression evaluates to an option value. The rules ADDF1 and ADDF2 propagate a `None` result in case the first or the second addend fails while evaluating. If the first argument evaluates to `Some` value n_1, the rule ADD1 evaluates the second expression to o_2, and then the result of the rule is the addition result of evaluating $n_1 + o_2$. Finally, ADD2 evaluates the concrete sum in the case also the second argument evaluates to `Some` value n_2. The division goes straightforwardly. It only details the case of the division by 0 in the DIV0 rule. If the second argument reduces to 0, `None` is propagated to the parent nodes in the derivation tree. In the original paper, all the `None` rules where bundled together, to reduce the number of the rules.

Similar to what we did with expression, we need to update the syntax of commands. We introduce $x :=_1 o$ as an intermediate stage for the assignment evaluation, $r;_1\ c_2$ for command sequencing, and `if1` for the `if-else` command. We do not have an intermediate stage for the `while` construct, as we do plan to evaluate it as we did in Section 1.2.1 by transforming the command into an `if`. The evaluation of the loop statement will happen following the big-step philosophy, meaning that its evaluation is contained in a single derivation tree. We present the new Backus-Naur-form syntax in Figure 1.12. Notice that we have introduced a new syntactic element, RES. This type defines the output sort of the new command transition relation $\Rightarrow_c$. We could have just kept going on using the option value. This change aims to mark the difference between expression and commands results. Then, in Figure 1.13, we present the pretty-big-step rules for this `while` language. Let us consider the assignment and the sequence rule set. For the assignment, the rule ASSIGN first evaluates the first expression to an option value o, then evaluates the new assignment $x :=_1\ o$ in σ. Notice that after the evaluation, the assignment operator is $:=_1$, a syntactic construct made for holding option

Store $\times$ CMD $\Rightarrow_c$ Res

$$\textsc{Skip}\ \frac{}{\sigma, \texttt{skip} \Rightarrow_c \sigma} \qquad \textsc{Assign}\ \frac{\sigma, e \Rightarrow_e o \quad \sigma, x :=_1\ o \Rightarrow_c r}{\sigma, x := e \Rightarrow_c r} \qquad \textsc{AssignN}\ \frac{}{\sigma, x :=_1\ \texttt{None} \Rightarrow_c \texttt{Exc}\ \sigma}$$

$$\textsc{AssignV}\ \frac{}{\sigma, x :=_1\ \texttt{Some}\ v \Rightarrow_c (\sigma + x \to v)} \qquad \textsc{Seq}\ \frac{\sigma, c_1 \Rightarrow_c r \quad \sigma, r\ ;_1\ c_2 \Rightarrow_c r}{\sigma, c_1; c_2 \Rightarrow_c r}$$

$$\textsc{SeqN}\ \frac{}{\sigma, \texttt{Exc}\ \sigma'\ ;_1\ c_2 \Rightarrow_c \texttt{Exc}\ \sigma'} \qquad \textsc{Seq1}\ \frac{\sigma', c_2 \Rightarrow_c r}{\sigma, \sigma'\ ;_1\ c_2 \Rightarrow_c r} \qquad \textsc{If}\ \frac{\sigma, e \Rightarrow_e o \quad \sigma, \texttt{if1}(o)\{c_1\}\texttt{else}\{c_2\} \Rightarrow_c r}{\sigma, \texttt{if}(e)\{c_1\}\texttt{else}\{c_2\} \Rightarrow_c r}$$

$$\textsc{IfN}\ \frac{}{\sigma, \texttt{if1}(\texttt{None})\{c_1\}\texttt{else}\{c_2\} \Rightarrow_c \texttt{Exc}\ \sigma} \qquad \textsc{IfT}\ \frac{\sigma, c_1 \Rightarrow_c r}{\sigma, \texttt{if1}(\texttt{Some}\ T)\{c_1\}\texttt{else}\{c_2\} \Rightarrow_c r}$$

$$\textsc{IfF}\ \frac{\sigma, c_2 \Rightarrow_c r}{\sigma, \texttt{if}(\texttt{Some}\ F)\{c_1\}\texttt{else}\{c_2\} \Rightarrow_c r} \qquad \textsc{While}\ \frac{\sigma, \texttt{if}(e)\{c; \texttt{while}(e)\{c\}\}\texttt{else}\{\texttt{skip}\} \Rightarrow_c r}{\sigma, \texttt{while}(e)\{c\} \Rightarrow_c r}$$

Figure 1.13: Commands Evaluation in Pretty-Big-Step

values instead of expressions, allowing for analyzing the expression evaluation result. Moreover, the result r of $\sigma, x :=_1\ o \Rightarrow_c$, is the overall result of the assignment. Then, if o is a `None` result, the overall result r is an exception `Exc` σ (rule AssignN). Otherwise, we update σ with the new binding. The result r is $(\sigma + x \to v)$. For the sequence command, the approach is quite similar. Similarly, we also have one more sequence construct $;_1$ for analyzing the intermediate results of evaluating the first sequenced command c_1. Notice that if the evaluation of c_1 is an exception, this is propagated (rule SeqN). Otherwise, r is a store σ'. The rule Seq1 evaluates in σ the syntactic construct $\sigma';_1 c_2$. The store σ' is the new sigma propagated by r, which has to be used for evaluating c_2. Indeed, σ' results from evaluating c_1 in σ, while σ is the initial store. This process of updating the store can be related to standard monadic techniques that propagate

implicitly updated information to some construct's continuation.

Chapter 2

The Skel Specification Language

Contents

« If you cannot get rid of the family skeleton,
you may as well make it dance. »
- George Bernard Shaw

Introduction

In the previous chapter, we presented different semantic styles for formally encoding the behavior of programming languages. In this chapter, we move the focus to the meta-languages used to capture semantics.

Section 2.1 informally introduces the meta-language Skel, a tiny functional specification language for writing Skeletal Semantics [8], by presenting, in Section 2.1.1, a minimal arithmetic language written in this language. Then, in the following section (Section 2.1.2), we broadly present `necro`, a set of tools designed for generating artifacts from a Skel mechanization. In the section, we overview the entire necro ecosystem, focusing on the generation of OCaml interpreters from a Skel. Afterward, in Section 2.1.3 we present an OCaml instantiation of an interpreter for the arithmetic language.

$$
\begin{array}{l}
\texttt{eval\_stmt}\ (\sigma, \texttt{While}\ (e, t)) \coloneqq \\
\quad \text{let}\, b = \texttt{eval\_expr}\ (\sigma, e)\, \text{in} \\
\quad \left(\begin{array}{l}
\text{let}\, \texttt{True} = b\, \text{in let}\, \sigma' = \texttt{eval\_stmt}\ (\sigma, t)\, \text{in} \\
\qquad\qquad\qquad \texttt{eval\_stmt}\ (\sigma', \texttt{While}\ (e, t)) \\
\text{let}\, \texttt{False} = b\, \text{in}\, \sigma
\end{array} \right)
\end{array}
$$

Figure 2.1: skeleton for the `While` constructor

We conclude the chapter by presenting an overview of related specification languages. The following chapters will discuss the limitations and differences between these works and Skel, accordingly to the purpose of the chapter.

This chapter contains introductory material [53, 41], which are not contributions of the author. Nevertheless, works presented in the thesis helped to refine the Skeletal Semantics formalism and the Skel language.

2.1 The Skel Language

Skeletal semantics is a *syntax* to define the semantics of programming languages in a concise yet powerful way, with a light formalism. This provides a way to easily manipulate the semantics, for instance to convert it into a Coq formalization or an OCaml interpreter. One of the strengths of Skeletal Semantics is the possibility to leave some constructions undefined, to let them be implementation dependent or for gradual specification. The theoretical concept behind Skeletal Semantics was presented in [8], and summarized in this thesis in the background chapter.

The “Skel” language, which has been defined to describe skeletal semantics, serves as support for the necro ecosystem [51], which provides, among other things, a generator of OCaml interpreters [42], a generator of gallina formalization [50], and a debugger generator [49].

The Skel formalization of the semantics of JavaScript internal algorithms presented in Chapter 4—which constitutes a main contribution of this thesis-has led us to propose several improvements to Skel, such as writing a functional meta-language provided with polymorphism and first-class functions.

Figure 2.1 shows an example of a skeleton for the evaluation of a *while* block of a vanilla while language in big-step style. We see all the main elements of Skeletal Semantics and we can observe that they are actually the main elements of any semantics.

- Recursion (`eval_stmt` and `eval_expr`) lets us define the semantics depending on the semantics of other terms (frequently subterms, but not always, as we can see in this example with the call `eval_stmt` $(\sigma', \texttt{While}\ (e, t))$).
- There are auxiliary functions and auxiliary types such as booleans, and possibility to match a variable against a given constructor (e.g `True`).
- The `let...in` construct is used to perform a sequence of operations.
- The branching (represented as a parenthesized system in Figure 2.1) is a choice between two possible rules. Often, the branches are mutually exclusive and a pattern matching at the start of the branch determines which branch is taken. Non-mutually exclusive branches also let us represent non-deterministic semantics (such as λ-calculus with no evaluation strategy).

2.1.1 An Arithmetic Language in Skel

We first present a simple arithmetic language in Skel. We keep the language simple, withouth introducing side effects, as the purpose of this section is to practically introduce, and show how to define semantics in Skel. Afterward, we will show how to generate an OCaml interpreter via `necroml` tool.

2.1.1.1 Syntax and Semantics

Consider the simple arithmetic language defined in Figure 2.2. We now describe its specification in Skel, depicted in Figure 2.3. First, the set $\mathbb{N}$ of natural numbers is required. As we do not want to specify this set in Skel, we simply state it exists using the *unspecified type declaration* `type nat`. Next, the syntax of terms needs to be given. To this end, we provide a *specified type declaration* in the form of an algebraic data type `term` with three constructors: `Const` taking a `nat`, and `Add` and `Sub` taking a tuple of two expressions.

Abstract syntax of the Arithmetic Language

$$\begin{aligned} \textsc{Nat}\ & n \in \mathbb{N} \\ \textsc{Expr}\ & e ::= n \mid (e_1 + e_2) \mid (e_1 - e_2) \\ \textsc{Value}\ & v ::= n \end{aligned}$$

Expr $\Downarrow$ Value

$$\frac{}{n \Downarrow n}\ \textsc{Const} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad \texttt{add}(n_1, n_2) = n}{(e_1 + e_2) \Downarrow n}\ \textsc{Add}$$

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad \texttt{sub}(n_1, n_2) = n}{(e_1 - e_2) \Downarrow n}\ \textsc{Sub}$$

Figure 2.2: Arithmetic Language

We now turn to the evaluation of this small language. First, we define the type of values as a specified type `value` with a single constructor `Nat` taking a `nat`. To define the evaluation of addition and subtraction expressions, we need to be able to actually compute them on integers. As integers are unspecified in our semantics, we assume the existence of *unspecified terms* `add` and `sub` by only giving their types. Finally, we can give the formalization of the $\Downarrow$ relation as the *specified term* `eval`, on the bottom of Figure 2.3. This specified term takes an expression `e` as input and it returns a value. As there are three rules to define the evaluation of arithmetic expressions, we correspondingly have three branches, introduced using the `branch ... or ... or ... end` construct. Each branch starts with a pattern matching using a let binding, such as `let Const n = e in ...`. This construct only computes if `e` has the shape `Const n` for some m, and if so the continuation of the let binding is evaluated, with `n` bound to m. If `e` does not have this shape, then the branch cannot be taken. The `let` construct is also used for sequencing. Let us consider the second branch. First, `e` is checked to have the shape `Add(e1,e2)`, then `e1` is recursively evaluated, and the result must be of the form `Nat n1`. Note that defined terms are by default recursive. Similarly, `e2` is recursively evaluated to `Nat n2`. Both numbers are added using the unspecified term

```
type nat

type expr =
| Const nat
| Add (expr,expr)
| Sub (expr,expr)

type value = | Nat nat

val add : (nat,nat) → nat
val sub : (nat,nat) → nat

val eval : (e:expr) : value =
    branch let Const n = e in Nat n
    or     let Add (e1,e2) = e in
           let Nat n1 = eval e1 in
           let Nat n2 = eval e2 in
           let n = add(n1,n2) in Nat n
    or     let Sub (e1,e2) = e in
           let Nat n1 = eval e1 in
           let Nat n2 = eval e2 in
           let n = sub(n1,n2) in Nat n
    end
```

Figure 2.3: Arithmetic Language Definition in Skel

`add` and the final result is returned, wrapped in the `Nat` constructor.

To sum up, a skel program is a list of type *declarations*, either *unspecified* or *specified* (by giving its constructors), and a list of term declarations, also either unspecified and containing only a name and type, or specified with an additional term. The possibility to declare non-specified types and terms is a really powerful tool. When defining a semantics, we sometimes do not want to go into details on how every type and every function works, or we want to delay giving the actual specification to a later iteration of the design of the semantics. Partial specifications allow for that.

In the next subsection we show practically how to instantiate the artifact generated by first presenting `necroml` tool, and afterwards showing how to produce a working interpreter.

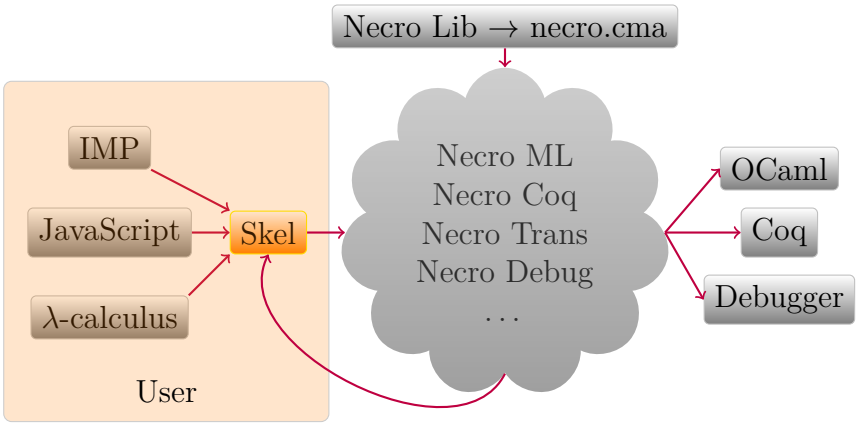

Figure 2.4: The Necro ecosystem.

2.1.2 The Necro Ecosystem

The Skel language by itself is a syntax for describing abstractly programming language semantics. The library `necrolib` and the set of tools `necrotrans`, `necroml`, `necrocoq`, and `necrodebug` are built for deriving machine-executable artifacts. In this section, we present both the library and the tools. We will focus more on the `necroml` because it is the main tool used in the thesis, as it allows us to extract an OCaml interpreter module. We show the `necro tools` pipeline in Figure 2.4.

`necrolib` It is the main library supporting the necro ecosystem. The `necrolib` generates a `.cma` OCaml library file, providing a parser, a typechecker, and a series of pretty printers to build and print the Skel abstract syntax tree (AST). The Skel language definition is small, consisting of almost 110 LOC. The size of the language allows us to easily define a suite of tools for manipulating Skel 's AST, generating artifacts in different languages.

`necrotrans` This tool is a part of the `necrolib` library. As the concrete Skeletal Semantics language is defined to support heavy use of monadic annotations, this tool is a transformer that, given a Skel file, expands all the binder-like function applications in the semantics.

`necroml` The tool is a generator of OCaml interpreters that, given a Skeletal Semantics `.sk` file, produces an OCaml functor representing its concrete semantics. The functor expects an implementation of unspecified types and terms. Once these are defined, the result is a working interpreter.

As the Skel language differs from OCaml, i.e., the `branch` is a non-deterministic construct representing different computation pathways, the produced artifact is a Skel shallow embedded in the OCaml's framework only for the unspecified types and terms. The behaviour of skeletons is defined by interpretation monads that declare how each skeleton is concretely computed. This choice fits with the theoretical framework itself, as the `branch` non-determinism allows to collect all the possible valid computations.

Executing the tool generates an OCaml file with 4 module types (`TYPES`, `MONAD`, `UNSPEC`, `INTERPRETER`) and two functors (`Unspec`, `MakeIntepreter`) to be implemented.

The module `TYPES` defines the names of all the unspecified types defined in the original `.sk` file. In this case, a user has to provide a module with a concrete type implementation.
The module `MONAD` represents the interpretation monad, explaining how the skeletons are concretely evaluated. Naïvely, a user can define a Skeletal Semantics identity monad. In Figure 2.5, on the top listing, we show the interpretation monad module type and its implementation on the bottom.
In this case, branches (`branch` function) are evaluated sequentially, considering the first successful branch as the branch to continue to evaluate. It is the closest approach to a shallow embedding in OCaml, separating the logics of Skeletal Semantics from the OCaml one. The problem with this implementation of the ID monad is that in case of future failures, it is impossible to backtrack and choose another branch if any. The `bind` and `apply` terms are pretty similar, but the first is the binder for the `let-in` skeleton, binding a fully evaluated term `t` to a variable `x`, in the environment of the continuation. The `apply` is the concrete term application used while evaluating the OCaml code. We can resume the differences with an example showing the generated code. Given a Skel snippet of code, `let r =@ f x in (*B*)`, where `f` is a higher-order term defined in the formalization of type `'a -> 'b`, representing a continuation, `x` a variable name or a formalization value of type `'a`, and `@` is a binder named "id_bind" defined in the skeletal semantics, the generated code can be represented compactly in the following listing.

```
module type MONAD = sig
  type 'a t
  val ret: 'a -> 'a t
  val bind: 'a t -> ('a -> 'b t) -> 'b t
  val branch: (unit -> 'a t) list -> 'a t
  val fail: string -> 'a t
  val apply: ('a -> 'b t) -> 'a -> 'b t
  val extract: 'a t -> 'a
end
```

```
module ID = struct
  exception Branch_fail of string
  type 'a t = 'a
  let ret x = x
  let bind x f = f x
  let rec branch l =
      begin match l with
      | [] -> raise (Branch_fail "no match")
      | bh::bt -> try bh () with Branch_fail _ -> branch bt
      end
  let fail s = raise (Branch_fail s)
  let apply f x = f x
  let extract x = x
end
```

Figure 2.5: The interpretation monad module and its identity monad instantiation

```
  let* _tmp = apply1 f x in
  apply2 id_bind _tmp (function r -> (*B*))
```

The operator `let*` and the functions `apply1` and `apply2` are respectively defined as:

```
let ( let* ) = M.bind
let apply1 = M.apply
let apply2 f arg1 arg2 =
  let* _tmp = apply1 f arg1 in
  apply1 _tmp arg2
```

The generated code can be presented extensively as showed below.

```
M.bind (M.apply f x)
       (λ _tmp -> M.bind (M.apply id_bind _tmp)
                         (λ _tmp1 -> M.apply _tmp1
                                             (function r -> (*B*))))
```

Notice that the result of `f x` is encapsulated in the skeletal semantics interpretation monad. So concretely, `f`'s signature is defined as `'a -> 'b t` as the skeletons evaluation is monadic. Hence term evaluation is encapsulated in skeletons monad. The variable `_tmp` is a temporary variable name holding a computation result of `f` applied to `x`.

The `fail` and `extract` terms respectively return an error message and extract values from the interpretation monad, usually the computation result. In the `necrolib` library, there are already some predefined interpretation monads, each providing Skeletal Semantics with a different meaning[1].

The module type `UNSPEC` contains all the types of the original `.sk` file, the unspecified terms signature, and the interpretation monad. The interpretation monad is present because it can influence the definition of unspecified terms.

Finally, the `INTERPRETER` module type defines the signatures of all the interpreter's types and terms. It is an extension of the module `UNSPEC`, containing the definition of the specified terms (the formalization).

Then, regarding the two functors, the `Unspec`, given an instantiation of the module `TYPES` and the interpretation monad instantiations as parameters, produces a concrete instantiation of the module `UNSPEC`. Notice that undefined unspecified terms return a `NotImplemented` result by default, and the actual implementation can overwrite their

1. Concrete implementation of the default interpretation monads in `https://gitlab.inria.fr/skeletons/necro-ml/-/blob/FSCD2022/necromonads.ml`

definitions. Then, the functor `MakeInterpreter` takes the instantiations of unspecified types and terms (the `Unspec` functor) and produces the interpreter for the formalization.

necrodebug The necro ecosystem also comprehends a debugger, very useful when writing big formalizations. The tool is an OCaml debugger generator, which given the Skel formalizations and a program that is written according to the formal syntax defined in the Skel file, it produces a step-by-step execution file using an abstract machine interpretation of Skel. The debugger provides two user interfaces, a terminal, and an HTML page generated by `js_of_ocaml`. An instantiation is needed to execute the debugger, similarly to the `necroml`. Actually, the same instantiation can be used both for the interpreter and the debugger. Then, the debugger needs to be extended by a suite of pretty printers.

necrocoq The `necrocoq` tool generate a deep embedding coq formalization from a Skel file. The produced artifact can be used to prove a program to be correct, or to validate properties of the programming language semantics itself, e.g. properties of the semantics of JavaScript.

2.1.3 An Interpreter for The Arithmetic Language

Once the skeletal semantics of the arithmetic language is defined, as presented in Figure 2.3 in a file `.sk`, an artifact can be produced with the `necroml` tool.

The `TYPES` module type This module is a list of declarations of unspecified types. In the Skel code, only the `nat` type has been left unspecified. We say that the naturals are the OCaml integers, defining the Skeletal type in the host language concrete definition of these numbers. We show the generated module and its instantiation in the listing below.

```
(* Generated module TYPES *)
module type TYPES = sig
  type nat
end

(* Custom implementation *)
module Types : sig
  type nat = int
end = Types
```

The `UNSPEC` module type This module is parametrized by a functor that provides default atomic definitions of the unspecified terms and the implementation of the specified terms and types. The `necroml` tool produces the following module type and functor.

In the top listing of Figure 2.6, the module `UNSPEC` presents the signature of the terms `add` and `sub`. In the central listing of Figure 2.6, we show the default functor `Unspec`. Each unspecified term has a default implementation that returns a `NotImplemented` result. In the bottom of Figure 2.6, we show the customized implementation of both these terms.

Notice that the interpretation monad, the `ID` monad shown in Figure 2.5, and the functor `Types` are included for instantiating the module `Unspec`. The `Types` module provides a concrete meaning to the unspecified type `nat`.

The `INTERPRETER` module type This module is a declaration of all the terms and types, either specified or non-specified. This module declares the signatures of all the interpreter components, and it is instantiated by the functor `MakeInterpreter`. The result of the generation is presented in Figure 2.7: on the top, we have the module type declaration, and on the bottom, the functor. The functor provides an implementation to the specified terms definition signatures defined on the left listing, a shallow embedding of the Skeletal Semantics language definition. Hence, by providing the `Unspec` functor as an argument to the `MakeInterpreter` one, one can produce a real interpreter for the arithmetic language.

```
module Interp = MakeInterpreter(Unspec)
```

```
(* Generated Module Type *)
module type UNSPEC = sig
  module M: MONAD
  include TYPES
  type value = | Nat of nat
  and expr = | Sub of (expr * expr) | Const of nat
  | Add of (expr * expr)
  val add: nat*nat ->nat M.t
  val sub: nat*nat ->nat M.t
end
```

```
(* Generated Default Functor *)
module Unspec (M: MONAD)(T: TYPES)=struct
  exception NotImplemented of string
  include T
  module M = M
  type value = | Nat of nat
  and expr = | Sub of (expr * expr) | Const of nat
  | Add of (expr * expr)
  let add _ = raise(NotImplemented "add")
  let sub _ = raise(NotImplemented "sub")
end
```

```
(* Custom Implementation *)
module Unspec = struct
  include Unspec(ID)(Types)
  let add ((a,b) : (nat*nat)) : nat M.t = M.ret (a + b)
  let sub ((a,b) : (nat*nat)) : nat M.t = M.ret (a - b)
end
```

Figure 2.6: The `UNSPEC` module type and its functors.

Then a program can be written and executed. For example, the program `1 + 1 - 2` can be encoded and executed by writing this code.

```
  open Unspec
  open Interp

  let _ = M.extract (eval (Sub (Add (Const 1,Const 1), Const (-2))))
```

```
(* Generated Module type *)
module type INTERPRETER =
sig
  module M: MONAD
  type nat
  type value = | Nat of nat
  and expr = | Sub of (expr * expr) | Const of nat
  | Add of (expr * expr)
  val add: nat*nat-> nat M.t
  val eval: expr-> value M.t
  val sub: nat*nat-> nat M.t
end
```

```
(* Generated Functor *)
module MakeInterpreter (U: UNSPEC)=
struct
  include U
  let ( let* ) = M.bind
  let apply1 = M.apply
  let rec eval e =
    begin match e with
    | Const n -> M.ret (Nat n)
    | Add (e1, e2) ->
        let* Nat n1 = apply1 eval e1 in
        let* Nat n2 = apply1 eval e2 in
        let* n = apply1 add (n1, n2) in
        M.ret (Nat n)
    | Sub (e1, e2) -> ...
    end
end
```

Figure 2.7: The `INTERPRETER` module type and its functor.

2.2 Related Specification Language

In the thesis, we present all the formalizations in Skel by presenting the behavior formally as an inference rule set or some algorithmic steps (in Chapter 4), then show how to encode them in our language.

To resume, Skel is the language for writing Skeletal Semantics, supported by `necro`, a library with a tool suite for manipulating the AST and producing different artifacts. We structure this section by presenting in each paragraph a different framework without

digging into which is better and why. We leave it to the following chapters. Generally, the evaluation we make is in terms of conciseness, closeness to the formal or informal definition of the language, and maintainability, which we think is crucial if we want to support work over the years when new releases of languages come out.

The properties of Skel rely upon the complexity of the language itself. It is a really tiny language. As shown in Chapter 1 a formal definition of a language defines the behavior of a language abstractly by defining premises that an evaluating configuration must satisfy. The operations held in the premises are usually not defined. For example, the addition relies on its natural meaning.
Considering the division we present in Figure 1.10, we explicitly write the number 0 in the `Div` rule, in its decimal form. Nevertheless, we could have chosen other ways to encode that number—hexadecimal, binary. A rule is just an abstraction of a behavior, decoupled from the concrete meaning of data.

In the Skel language, we can write partial specifications by defining unspecified types and terms without actual implementation.
Moreover, the language does not have native types except for user-defined constructed types. The unspecified types allow for being abstract enough not to sacrifice the readability of the semantics for constraints dependent on types. Indeed, it can be as abstract as an inference rule by only declaring unspecified components of the semantics. Then, the language is tiny, making the language prone to be used for generating artifacts of different nature. The Skel language definition stays in a couple of hundreds of lines.

In the following paragraphs, we briefly introduce the frameworks related to the thesis' contribution.

Lem Lem [55, 47] is a specification language allowing to extract from the source file an OCaml interpreter and mechanizations in a proof assistant framework, namely Coq and Isabelle/HOL. The scope of this specification language is vast, spreading from communication protocols to weak memory models. The language itself is complex, allowing complex type inference and constructed modules, making the overall language quite heavy to manipulate. The language has native types, such as four number types, and strings.

Ott Ott [66] is a tool for specifying programming languages, allowing to extract from an Ott source file a LaTeX representation, Isabelle/HOL, and Coq files, and an exe-

cutable OCaml file. It also provides ways to derive a Lem mechanization. This tool is known for its writing style, which is close to the handwritten inference rules.

K K [63] is a formal framework for writing executable semantics in a rewriting style. The framework is designed to define modular semantics in its components, such as the state. The K framework has a steep learning curve, making it quite difficult to approach. Nevertheless, many applications came from this framework, for example formalizations of JavaScript [56], C [21, 28], Java [10], and PHP [23]. Some work [17] recently provided methods for proof generation and checking in Isabelle/HOL, but, to our knowledge, no work yet provides proofs of meta-theories of a K formalization. Nevertheless, the K framework is powerful, and offers tools for analyzing K programs, such as a symbolic executor, a test-case generator, a runtime monitor, and an interpreter. This semantic framework has a Isabelle/HOL mechanization of the semantics, to formally show undesirable behaviours in the current implementations of the language [40].

2.3 Conclusion

In this chapter, we first presented features of the Skel language by implementing the Skeletal Semantics of a simple arithmetic language. The language can be put in the same cathegory of semantic frameworks with Ott, Lem, and K, with the difference that Skel comes with a simple syntax and with no a priori semantics, allowing to manipulate easily a Skel AST and producing different artifacts. Afterward, we introduced tools for generating different artifacts from a Skel source. We focused more on the `necroml` tool as the thesis relates more with concrete interpretation rather than formal verification of meta-theories of a formalization. Hence, in Section 2.1.3 we showed how to generate an OCaml interpreter module of the arithmetic language, and how to instantiate it for obtaining a working interpreter.

Contributions in this thesis pushed the language to move toward a functional path. For example, representing JavaScript semantics—see Chapter 4-needed something more from the language, namely polymorphism and higher-order terms as first-class language citizens, to capture concisely and faithfully the ECMAScript specification algorithms. Skel development has been strongly influenced by the formalizations presented in the thesis.

Chapters 1 and 2 presented ways to write semantics, either on paper or on the machine, and offer the necessary knowledge to appreciate the contributions of this manuscript. The following chapters will exploit some Skel features for writing concise, modular, and executable semantics.

Part II

Effectful Semantics

Chapter 3

Describing Concisely Effectful Semantics

Contents

« Je n'ai fait cette lettre-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte. »
- Blaise Pascal [60]

Introduction

Writing a formal definition of a programming language specification helps state precisely the behavior of the language constructs and also derive frameworks in which it is possible to prove programs and/or language properties. The result of this task must ensure that the actual semantics of the language is captured.

To ensure that a work written in a semantic framework adequately describes the meaning of the programming language, one must preserve the following properties:

1. The formalization should be textually close to the specification of the language if it exists.

2. An executable formalization should be tested against the test suite provided by the language. Thus, a suitable tool to describe semantics should make it easy to stay close to language definitions, be they algorithmic or based on inference rules, and the tool should provide a straightforward way to derive an executable version of the semantics and run code, including tests.

3. Language definitions written in such a tool should be easy to manipulate, for instance, to derive a mechanized version in a proof assistant, inspect the semantics, or transform it.

In this chapter, we define a method for structuring the semantics of a sequence toy languages written in Skel. In Section 3.1, we extend the simple arithmetic language presented in Section 2.1.1 by introducing the division and multiplication expressions, a try-catch expression, and a datatype describing the exceptions of the language. We then structure the chapter in the following way. Section 3.2 extends the effectful arithmetic language by introducing first-class functions and a fix-point operator. In Section 3.3, we continue extending this language by adding a mutable state and extending the language with some state operators. In Section 3.5, we show how to add delimited computations and explicitly handle the natural control flow of a program. We show in each section how to introduce these new features without disrupting the formalization of the constructs that do not directly use them. The final language we present is syntactically simple, but its behavior is far from trivial. We also provide the semantics and the interpreter of every language presented in the chapter in a source repository[1]. Finally, in Section 3.6, we relate our approach to existing work.

1. `https://gitlab.inria.fr/akhayam/programming_skel`

Contribution

The chapter presents:

- a systematic approach to capture multiple side-effects in Skel formalizations.
- exploit the power of monads to write readable, and maintainable semantics in Skel.
- structurally solid evaluation functions. Extending a language by introducing new effectful operations does not impact the code describing the previously written core language.

3.1 Effectful Arithmetic Language

In this section, we first show how to introduce effects in Skeletal Semantics by adding the division operator to the arithmetic language presented in Section 2.1.1. The evaluation of an expression with a division can fail because of a division by 0 exception. We first try to handle effects explicitly, and then we show how a monadic approach helps to structure the language's formalization. Handling explicitly side effects is transparent but not scalable when writing a real-world programming language formalization.

If we add division to the language, a program may try to perform a division by zero. To capture this information, we introduce an exception type. This type is defined by two constructors, `Exc`, representing that the computation has failed, and `Ok`, carrying type `value`, which means that the computation is successfully calculated. This type is similar to **OCaml**'s `option` type or **Haskell**'s `maybe` type. We show the syntax definition of the exception type in BNF and Skel in Figure 3.1. We put in the syntax `...` for saying that there are other type constructors defined for the expression type, namely `Const`, `Add`, and `Sub`.

We present the evaluation rules of this language, inference rules, and Skel in Figure 3.2 and Figure 3.3. We omit the rules for subtraction and multiplication.
First, we give an alias `type output := exc` such that the result of `eval` is always `output`, even for more complex languages (we will change the alias when needed). We

Formal Syntax

$$
\begin{array}{rcl}
\textsc{Exc}\ o & ::= & \mathit{Exc} \\
 & | & \mathit{Ok}\ v \\
\textsc{Expr}\ e & ::= & \ldots \\
 & | & (e_1/e_2) \\
 & | & \texttt{try}\{e_1\}\{e_2\}
\end{array}
$$

```
type exc  = | Exc
            | Ok value
type expr = | ...
            | Div (expr,expr)
            | Try (expr,expr)
```

Figure 3.1: Syntax of the Exception type

$e \Downarrow o$

$$\textsc{Const}\ \frac{}{n \Downarrow \mathit{Ok}\ n} \qquad \textsc{Add}\ \frac{e_1 \Downarrow \mathit{Ok}\ n_1 \quad e_2 \Downarrow \mathit{Ok}\ n_2 \quad \texttt{add}(n_1, n_2) = n}{(e_1 + e_2) \Downarrow \mathit{Ok}\ n}$$

$$\textsc{AddF1}\ \frac{e_1 \Downarrow \mathit{Exc}}{(e_1 + e_2) \Downarrow \mathit{Exc}} \qquad \textsc{AddF2}\ \frac{e_1 \Downarrow \mathit{Ok}\ n_1 \quad e_2 \Downarrow \mathit{Exc}}{(e_1 + e_2) \Downarrow \mathit{Exc}}$$

$$\textsc{Div}\ \frac{e_1 \Downarrow \mathit{Ok}\ n_1 \quad e_2 \Downarrow \mathit{Ok}\ n_2 \quad n_2 \neq 0 \quad \texttt{div}(n_1, n_2) = n}{(e_1/e_2) \Downarrow \mathit{Ok}\ n} \qquad \textsc{Div0}\ \frac{e_1 \Downarrow \mathit{Ok}\ n_1 \quad e_2 \Downarrow \mathit{Ok}\ n_2 \quad n_2 = 0}{(e_1/e_2) \Downarrow \mathit{Exc}}$$

$$\textsc{DivF1}\ \frac{e_1 \Downarrow \mathit{Exc}}{(e_1/e_2) \Downarrow \mathit{Exc}} \qquad \textsc{DivF2}\ \frac{e_1 \Downarrow \mathit{Ok}\ n_1 \quad e_2 \Downarrow \mathit{Exc}}{(e_1/e_2) \Downarrow \mathit{Exc}}$$

$$\textsc{TryF1}\ \frac{e_1 \Downarrow \mathit{Exc} \quad e_2 \Downarrow r_2}{\texttt{try}\{e_1\}\{e_2\} \Downarrow r_2} \qquad \textsc{Try}\ \frac{e_1 \Downarrow \mathit{Ok}\ v}{\texttt{try}\{e_1\}\{e_2\} \Downarrow \mathit{Ok}\ v}$$

Figure 3.2: Arithmetic Language Rules Extended with Division and Exceptions

assume two unspecified terms to test for zero and not zero and an unspecified third term to divide two numbers. We also assume that this last unspecified term should never be called with a divisor which is zero. We finally extend the syntax of arithmetic expressions with a division operator and a `try with` construct to catch exceptions.

As there are 13 rules to evaluate arithmetic expressions with division (the three rules

```
type output := exc
val zero: nat → ()
val not_zero: nat → ()
val div: (nat,nat) → nat
val mul: (nat,nat) → nat
val eval (e: expr) : output =
    branch let Const n = e in (Ok (Nat n))
    or     let Add (e1,e2) = e in
           let o1 = eval e1 in
           branch let Exc = o1 in o1
           or     let Ok (Nat n1) = o1 in
                  let o2 = eval e2 in
                  branch let Exc = o2 in o2
                  or     let Ok (Nat  n2) = o2 in
                         let n = add (n1,n2) in
                         Ok (Nat n)
                  end
           end
    or     let Div (e1,e2) = e in
           let o1 = eval e1 in
           branch let Exc = o1 in o1
           or     let Ok (Nat n1) = o1 in
                  let o2 = eval e2 in
                  branch let Exc = o2 in o2
                  or     let Ok (Nat  n2) = o2 in
                         branch zero n2; Exc
                         or     not_zero n2;
                                let n = div(n1,n2) in
                                Ok (Nat n)
                         end
                  end
           end
    or     let Try (e1,e2) = e in
           let o1 = eval e1 in
           branch let Exc = o1 in eval e2
           or     let Ok (Nat r1) = o1 in o1
           end
    or     ...
    end
```

Figure 3.3: Arithmetic Language in Skel Extended with Division and Exceptions

for subtraction being omitted), one might expect 13 branches to be needed. Fortunately, Skel lets us factorize some of the behavior. Consider the case of addition: in the three rules shown, the first sub-expression is always evaluated. Hence the corresponding Skel rule also evaluates it to o_1, and then it branches for inspection. If it is an exception, then it is propagated. Otherwise, it is a number n_1, and the computation may continue with the evaluation of e_2. Such factorization may be compared to the one obtained using Pretty-Big Step Semantics [15], as shown in Section 1.4.

This description is still somewhat naïve because the same code structure must be repeated for division (in blue) and subtraction (not shown). Before improving the implementation, let us describe the code for division and `Try`. Regarding division, once both sub-expressions are successfully computed, the second one is tested against zero: if it is zero, an exception is raised; if it is not, the division may proceed. Once again, branches are used to capture a conditional. As branches are inherently non-deterministic, one must ensure that the required conditions are fully stated. Note that we use the syntactic sugar `foo; rest` for `let _ = foo in rest`, and rely on the partiality of the `zero` and `not_zero` unspecified terms.

Regarding the `Try` term, we first evaluate its first sub-expression. If it results in an exception, we evaluate the second sub-expression; if it succeeds, we return its value.

One may observe that adding exceptions makes the code significantly bigger and less readable, with much more of structural duplication.

To capture such effects in Skel, we rely on the classical approach of monads. As a first approximation, a monad is a triple `{M, return, bind}`, where

- M is a monadic type constructor, taking a type and returning a type;
- `return` : $\alpha \to M\alpha$ is a function that takes a pure value of type α and returns the same value in the monad;
- `bind` : $M\alpha \to (\alpha \to M\beta) \to M\beta$ is a function that takes w, a monadic value of type α, a continuation f of type $\alpha \to M\beta$, and `bind w f` returns a new monadic value of type $M\beta$.

Monads also come with laws that state that `ret` and `bind` are inverse of each other in some sense. These laws can be ignored because we do not prove the properties of our code.

To write monads in Skel, we introduced‘[52] higher-order terms as first-class citizens as well as parametric polymorphism. Polymorphism is always explicit: the polymorphic

type `exc<a>` is parametric in its type parameter `a`, which can be used in its definition (as in `type exc<a> = | Exc | Ok a`). Polymorphic term declarations and polymorphic constructors also have this typesetting. Examples are the `return` and `bind` defined below.

```
val return<a> (v:a) : exc<a> = Ok<a> v

val bind<a,b> (w: exc<a>) (f: a → exc<b>) : exc<b> =
  branch let Exc = w in throw<b> ()
  or     let Ok v = w in f v
  end
```

On the other hand, constructors used as patterns do not require polymorphic annotations as they can be locally inferred.

These three declarations actually form the exception monad that we use in Figure 3.4. They include an additional return always to throw and functions specialized to the value type (to avoid annotations). We do not include the `return` and the `bind` function in the figure.

In the same figure, below, we give the `eval` function using the exception monad. As the `bind` function expects a function as a second argument, and as we prefer to avoid defining such functions as top-level terms, we introduce a final part of Skel 's syntax: anonymous functions. They are simply written `λp:t → s`, where `p` is a pattern, `t` is a type, and `s` is a skeleton. We thus give as a continuation of each bind the remainder of the code, and there is no need to branch on the status of an evaluation to decide whether to continue or throw, as the `bind` takes care of it.

To make the code more readable, we make the use of `bind` for continuation code more straightforward. we declare that the higher-order term is a bind and assign it a symbol by writing `binder @ := bind`. When this is done, code like `bind o (λ(x:t) → s)` can simply be written `let x =@ o in s` (type annotations are inferred). The resulting code is shown in Figure 3.5. We can even abuse this notation to define another bind-like function `exccont`, which calls its continuation in the case an exception was raised and otherwise return the given value to give it a symbol and use it in the code for `Try`. We associate this behavior with the symbol `?`, which represents the monadic term `exccont`. If this code, the construct `e1; e2`, stands for `let _ = e1 in e2`, then `eval e1;? eval e2` states evaluating `e1`, if it returns an exception, then evaluate `e2`,

```
val throw<a> (_:()) : exc<a> = Exc<a>

type output := exc<value>

val ret (v:value) : output = return<value> v
val thr (_:()) : output = throw<value> ()

val eval : (e:expr) → output
    branch let Const n = e in ret (Nat n)
    or     let Div (e1,e2) = e in
           let o1 = eval e1 in
           bind<value,value> o1 (λv1:value →
               let Nat n1 = v1 in
               let o2 = eval e2 in
               bind<value,value> o2 (λv2:value →
                   let Nat n2 = v2 in
                   branch let _ = zero n2 in
                          thr ()
                   or     not_zero n2;
                          let res = div(n1,n2) in
                          ret n
                   end))
    or     let Add (e1,e2) = e in
           let o1 = eval e1 in
           bind<value,value> o1 (λv1:value →
               let Nat n1 = v1 in
               let o2 = eval e2 in
               bind<value,value> o2 (λv2:value →
                   let Nat n2 = v2 in
                   let n = add(n1,n2) in
                   ret n))
    or     ...
    end
```

Figure 3.4: Exception monad in Skel, and its application to the `eval` function

otherwise return its result. Introducing a symbol for a bind operator is not required; one may directly use the notation `let x =%bind o in s` to stand for `bind o (λx → s)`. In the following listing, we show the implementation of the function `exccont`.

```
val eval (e: expr) : output
    branch let Const n = e in ret (Nat n)
    or     let Div (e1,e2) = e in
           let Nat n1 =@ eval e1 in
           let Nat n2 =@ eval e2 in
           branch zero n2; thr ()
           or     not_zero n2;
                  let n = div(n1,n2) in
                  ret (Nat n)
           end
    or     let Try (e1,e2) = e in
           eval e1;?
           eval e2
    or     let Add (e1,e2) = e in
           let Nat n1 =@ eval e1 in
           let Nat n2 =@ eval e2 in
           let n = add(n1,n2) in
           ret (Nat n)
    or     ..
    end
```

Figure 3.5: Exception Monad and its Symbolic Application to `eval`

```
val exccont<a> (v:exc<a>) (f:() → exc<a>): exc<a> =
    branch let Exc = v in f ()
    or     let Ok v' = v in
           return<a> v'
    end

binder ? := exccont
binder @ := bind
```

In this subsection, we showed how to extend a language in Skel, introduced monads and binders, and used them. We defined a writing pattern that makes the semantics highly readable and compact.

```
let zero (a:nat) : unit M.t =
match a with
| 0 -> M.ret ()
| _ -> M.fail "not zero"
```

Figure 3.6: The `zero` term implementation

Instantiation

To instantiate an interpreter to the language presented in this section, a user can simply, after a `necroml` run, extend or include the `Unspec` functor shown in Figure 2.6. We provide an implementation to the unspecified terms `zero`, `not_zero`, `div`, and `mul`. Note that the `zero` and the `not_zero` terms are partial functions. These function may fail in the Skel interpretation monad. We show the implementation of the `zero` term in Figure 3.6.

Remark

We remark that the monads we present in this semantics are not related to the interpretation monads of the ml-instantiation. The monadic approach we present represents the way we write semantics.

3.2 PCF

Even with exceptions, a language for arithmetic expressions is still far from an actual programming language. We thus extend it with variables, first-class functions, a sequencing operator, skip, and a fix-point operator.

Formal Syntax

$$
\begin{aligned}
&\ldots \\
\textsc{Env}\ \ &\sigma : \textsc{Var} \to \textsc{Value} \\
\textsc{Var}\ \ &x \in \Sigma^{+} \\
\textsc{Value}\ \ &v ::= n \\
&\quad \mid (x, t_x, \sigma_{cl}) \\
&\quad \mid (x, t_x, \sigma_{cl}, x_f) \\
&\quad \mid () \\
\textsc{Expr}\ \ &e ::= \ldots \\
&\quad \mid x \\
&\quad \mid \lambda\, x.e_x \\
&\quad \mid e_1\ e_2 \\
&\quad \mid \texttt{let}\ x = e_x\ \texttt{in}\ e_c \\
&\quad \mid \texttt{if0}(e_c)\{e_1\}\{e_2\} \\
&\quad \mid \texttt{fix}\ f x.e_x \mid e_1; e_2 \\
&\quad \mid \texttt{skip}
\end{aligned}
$$

```
...
type env
type var
type value =
| Nat nat
| Clos (var,expr,env)
| ClosRec (var,expr,env,var)
| Unit
type expr =
| ...
| Var var
| Lam (var,expr)
| App (expr,expr)
| LetIn (var,expr,expr)
| If0 (expr,expr,expr)
| Fix (var,var,expr)
| Seq (expr,expr)
| Skip
```

The evaluation of this language requires an environment σ : VAR $\to$ VALUE that maps variables to values. We define the semantic judgement as $\sigma, e \Downarrow$ `Exc Value`. In the arithmetic language, `Nat` was the only possible value. In PCF, we add the *unit* value and *closures*. The value *unit* represents is a singleton value we use to return no value. Then, we define two types of closures. The first type represents the result of evaluating a lambda expression, and this value stores the bound variable `x`, the body of the abstraction `eb`, and the current environment `s` as `Clos (x,eb,s)`. The second type of closure represents recursive closed expressions. Similarly to the non-recursive closure, the recursive one stores `x`, `eb`, `s`, and the name of the recursive expression `f` as `ClosRec (x,eb,s,f)`. The environment must specify which values the free variables of `eb` are bound to.

The presence of an environment to evaluate an expression means that the `eval` function must change its signature: it now takes both an environment and an expres-

sion.The output of this function is a value in the exception monad. The evaluation is shown in both Figure 3.7 and Figure 3.8. The case of arithmetic operations is un-

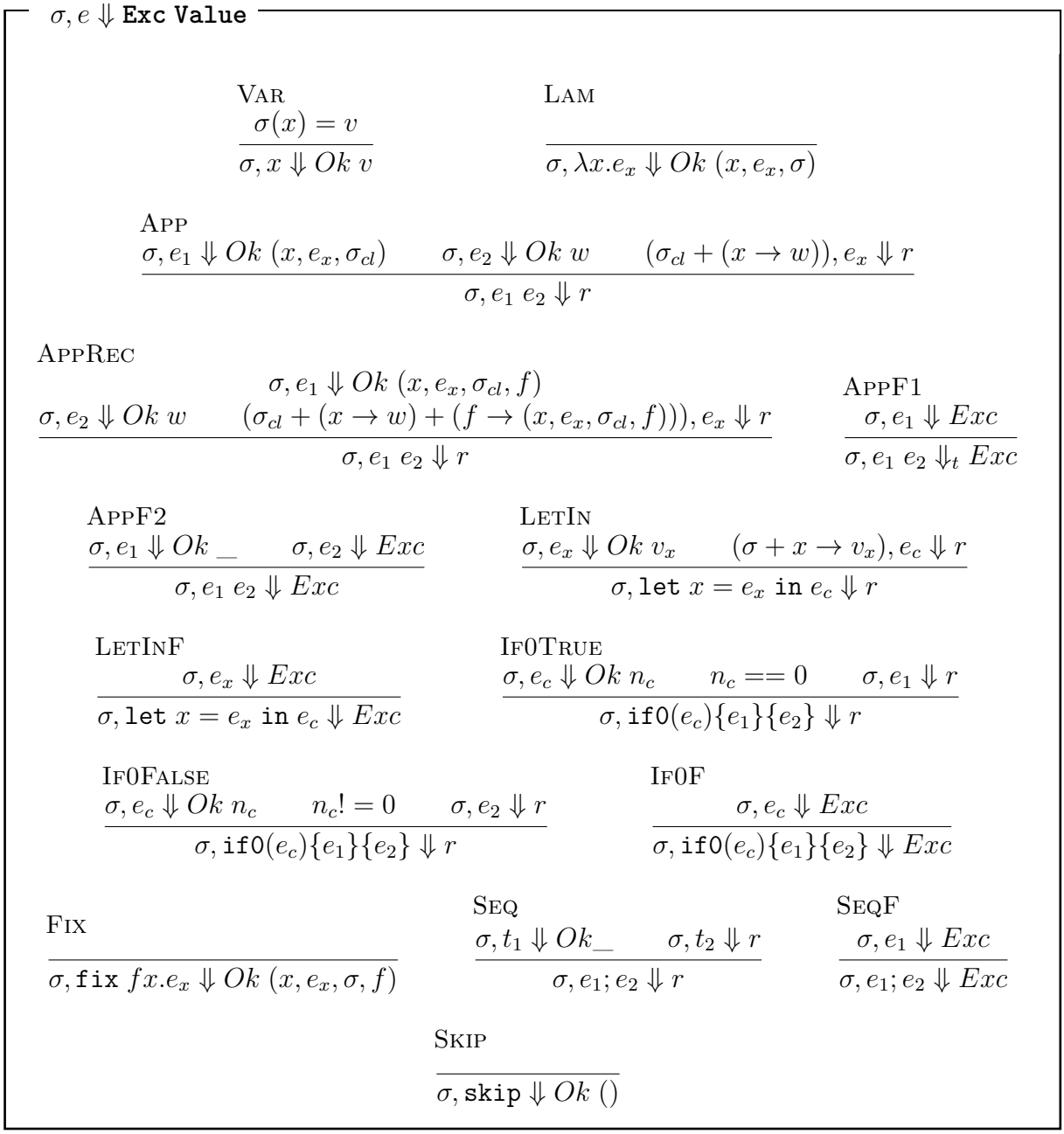

Figure 3.7: PCF Semantic Rules

changed, except for adding the environment to the recursive call of `eval`. Variables are looked up in the environment. The evaluation of an application `App(e1,e2)` in the en-

```
val eval (s:env) (e:expr) : output =
 branch ...
 or     let Add (e1,e2) = e in
        let Nat n1 =@ eval s e1 in
        let Nat n2 =@ eval s e2 in
        let n = add(n1,n2) in ret (Nat n)
 or     let Var x = e in
        let v = getEnv s x in ret v
 or     let Lam (x,ex) = e in
        ret (Clos (x,ex,s))
 or     let App (e1,e2) = e in
        let cl =@ eval s e1 in
        let w =@ eval s e2 in
        branch let Clos (x,eb,sc) = cl in
               let sc' = extEnv sc x w in
               eval sc' eb
        or     let ClosRec (x,eb,sc,f) = cl in
               let sc' = extEnv sc x w in
               let sc'' = extEnv sc' f cl in
               eval sc'' eb
        end
 or     let LetIn (x,ex,ec) = e in
        let vx =@ eval s ex in
        let s' = extEnv s x vx in
        eval s' ec
 or     let If0 (ec,e1,e2) = e in
        let Nat nc =@ eval s ec in
        branch zero nc; eval s e1
        or     not_zero nc; eval s e2
        end
 or     let Fix (f,x,ex) = e in
        ret (ClosRec (x,ex,s,f))
 or     let Seq (e1,e2) = e in
        eval s e1 ;@
        eval s e2
 or     let Skip = e in ret Unit
 end
```

Figure 3.8: PCF Semantics in Skel

```
let extEnv (e:env) =
    M.ret (fun x -> M.ret (fun v -> M.ret (SMap.add x v e)))
```

Figure 3.9: `extEnv` instantiation in OCaml

vironment `s` first evaluates `e1` in `s`. The result must be a closure `Clos(x,ex,sc)` or a recursive closure `ClosRec(x,ex,sc,f)`. The argument `e2` is then evaluated in `s` to `w`. Finally, the body of the closure `ex` is evaluated in the environment `sc` extended with a mapping from `x` to `v`. In the recursive case, `sc` is also extended with a binding from `f` to `ClosRec(x,ex,sc,f)`. Note that these steps are done in the exception monad: once one evaluation returns an exception, it is propagated, and the evaluation is interrupted.

We assume that all the arithmetic operations, including multiplication, are described by our semantics, so in the figure, we show only the evaluation steps for the `Add` constructor. We use integers for conditionals with the `If0` operator.

To access and extend the (unspecified) environment, we assume the existence of the `getEnv` and `extEnv` terms. The environment is assumed to be a partial mapping from VAR to VALUE. In case a mapping does not exist, the result of getting a variable's value is a Skel failure. Indeed, we decided not to capture it in the exception monad to be consistent with other errors, such as applying a number as a function. These errors could be detected statically by writing a PCF type checker.

Figure 3.8 shows the new `eval` function for PCF in Skel. The evaluation of the language is almost standard. The rules follow the ones from [62] with two exceptions: we use an environment instead of substitution and provide a big-step semantics. Because of the former, the `Fix` operator's semantics is described as follows. `Fix (f,x,ex)` is evaluated to a recursive closure `ClosRec (x,ex,sc,f)`, which is a value. The extension of the environment with f bound to the recursive closure is delegated to the application if it happens. Notice that this evaluation is close to the evaluation of a λ-expression. A PCF program defining and using factorial can be written as follows, in concrete syntax (in Skel, we would need to use the constructors `LetIn`, `Fix`, etc.)

```
let fact = fix factrec n. if0(n) {1} {n*(factrec (n-1))} in
fact 42
```

Instantiation

To generate an interpreter, one must extend the `Types` functor providing a concrete type to variables and the environment σ. Moreover, The `Unspec` functor needs a definition for the functions `getEnv` and `extEnv`. We provide a demonstrative implementation of the `extEnv` in Figure 3.9. The `SMap` functor represents the library to manipulate maps from strings to values. The functor is constructed with the following code, `module SMap = Map.Make(String)`, and the `type env` is defined as `Unspec(M)(Types).value SMap.t`, a map from strings to the language value type defined in Skel.

3.3 Adding State to the PCF language

We now extend our language with a state and references. Allocating a new reference with the operator $\&v$ consists of extending a state μ with a new memory location l containing an initial value v. We write the semantic judgement a $\mu, \sigma, e \Downarrow (\texttt{Exc Value}, \mu)$. We define two other operations, $!l$ for accessing the contents of a location l (*dereferencing*), and the infix operator $l := v$ for setting a new value v in an existing allocated location l. Formally, we can extend the syntax as follows.

Formal Syntax

$$
\begin{aligned}
&\ldots \\
&\textsc{Loc}\ l \\
&\textsc{State}\ \mu : \textsc{Loc} \to \textsc{Value} \\
&\textsc{Value}\ v ::= \cdots \mid l \mid () \\
&\textsc{St}(\alpha)\ st ::= \mu \to (\alpha, \mu) \\
&\textsc{Expr}\ e ::= \ldots \mid \&e \mid !e \mid e_1 := e_2
\end{aligned}
$$

```
...
type loc
type value = | ...
             | Loc loc
             | Unit
type state
type st<a> := state → (a,state)
type expr = | ... | Ref expr
            | Get expr
            | Set (expr,expr)
```

In Skel, we extend the `value` type with a `Loc` constructor for location values and

`Unit`. We leave the `loc` and `state` types unspecified and define the monadic state type (`type st<a>`). Finally, we extend `expr` type with three new language constructors: `Ref` for &, `Get` for !, and `Set` for the infix operator := .

To describe the semantics, we extend our monad to include a state monad. A computation in a state monad is a function that expects the current value of the state to return its result alongside the updated state. We first define the state bind operator `stbind`, denoted with `@m`. This binder is the regular state binder.

```
val stbind<a,b> (v:st<a>)(f:(a → st<b>)) : st<b> =
    λ s : state → let (v',s') = v s in f v' s'
binder @m := stbind
```

Next, we combine the state and exception monad, writing its `bind` operator, denoted with `@`. The monadic type considered is `v:st<exc<a>>`. It means that, given a state μ, the application `v` μ returns a tuple `(v', μ') : (exc<α>, state)`. We keep exceptions inside the state as a computation may mutate the state before throwing an exception, and this mutated state needs to be propagated. We show the new `bind` in the following snippet of code.

Remark

We chose to combine the state monad and the exception monad in this order to guarantee the state's persistence in case of a failure. In the example, we are interested in cumulating results, always being able, in case, to inspect the state. We define the combined monadic type as STATE $\rightarrow$ (EXC a, STATE). Combining in the other way means having the following signature: EXC(STATE $\rightarrow$ (a, STATE)). In case of an exception, the computation loses the most updated state.

```
val bind<a,b> (v:st<exc<a>>)(f:(a → st<exc<b>>)) : st<exc<b>> =
    let v' =@m v in
    branch let Exc = v' in throw<b> ()
    or     let Ok v'' = v' in f v''
    end
binder @ := bind
```

We update the `exccont` binder, represented as `?`, to take the monad into account. This lets us keep the `Try` case of the semantics unmodified. The update is symmetric to what did with the monadic binder.

```
val exccont<a> (v:st<exc<a>>)(f:(() → st<exc<a>>)) : st<exc<a>> =
    let v' =@m v in
    branch let Exc = v' in f ()
    or     let Ok v'' = v' in return<a> v''
    end
binder ? := exccont
```

Finally, we redefine the `return` and the `throw` term. Here again, we inject the `exc<a>` return type into the state monad.

```
val stret<a> (v:a) : st<a> = λ s:state → (v,s)
val return<a> (v:a) : st<exc<a>> = stret<exc<a>> (Ok<a> v)
val throw<a> (_:()) : st<exc<a>> = stret<exc<a>> Exc<a>
val ret (v:value) : output = return<value> v
val thr (_:()) : output = throw<value> ()
```

In Figure 3.10, we show the semantics of `Ref`, `Get`, and `Set` in Skel and big-step. We define three concrete atomic operations on memory: `alloc` is a function for allocating a new memory location with an initial value, `m_r` returns the value bound to a reference, and `m_w` writes a value in a location. The specified terms `newref`, `get`, and `set` are the monadic versions of `alloc`, `m_r`, and `m_w`. We show them below.

$\mu, \sigma, e \Downarrow (\textbf{Exc Value}, \mu)$

$$\textsc{Ref}\ \frac{\mu, \sigma, e_r \Downarrow (Ok\ v, \mu'') \qquad \texttt{alloc}(\mu'', v) = (l, \mu')}{\mu, \sigma, \&e_r \Downarrow (Ok\ l, \mu')} \qquad \textsc{RefF}\ \frac{\mu, \sigma, e_r \Downarrow (Exc, \mu')}{\mu, \sigma, \&e_r \Downarrow (Exc, \mu')}$$

$$\textsc{Get}\ \frac{\mu, \sigma, e_g \Downarrow (Ok\ l, \mu') \qquad \mu'(l) = v}{\mu, \sigma, !e_g \Downarrow (Ok\ v, \mu')} \qquad \textsc{GetF}\ \frac{\mu, \sigma, e_g \Downarrow (Exc, \mu')}{\mu, \sigma, !e_g \Downarrow (Exc, \mu')}$$

$$\textsc{Set}\ \frac{\mu, \sigma, e_l \Downarrow (Ok\ l, \mu'') \qquad \mu'', \sigma, e_v \Downarrow (Ok\ v, \mu''') \qquad \mu'''(l, v) = \mu'}{\mu, \sigma, e_l := e_v \Downarrow (Ok\ (), \mu')}$$

$$\textsc{SetF1}\ \frac{\mu, \sigma, e_l \Downarrow (Exc, \mu')}{\mu, \sigma, e_l := e_v \Downarrow (Exc, \mu')} \qquad \textsc{SetF2}\ \frac{\mu, \sigma, e_l \Downarrow (Ok\ l, \mu'') \qquad \mu'', \sigma, e_v \Downarrow (Exc, \mu')}{\mu, \sigma, e_l := e_v \Downarrow (Exc, \mu')}$$

```
val output := st<exc<value>>
val eval (s:env)(e:expr) : output =
    branch ...
    or      let Add (e1,e2) = e in
            let Nat n1 =@ eval s e1 in
            let Nat n2 =@ eval s e2 in
            let n = add(n1,n2) in  ret (Nat n)
    or      let Ref er = e in
            let v =@ eval s er in
            newref(v)
    or      let Get eg = e in
            let Loc l =@ eval s eg in
            get(l)
    or      let Set (el,ev) = e in
            let Loc l =@ eval s el in
            let v =@ eval s ev in
            set(l,v)
    end
```

Figure 3.10: PCF with References

```
val alloc: value → st<loc>
val m_r: loc → st<value>
val m_w: (loc, value) → st<()>
val newref (v:value) : output =
    let l =@m alloc v in ret (Loc l)
val get (l:loc) : output =
    let r =@m m_r l in ret r
val set ((l,v):(loc,value)) : output =
    m_w(l,v);@m ret Unit
```

Note that the three concrete functions `alloc`, `m_r`, and `m_w` are left unspecified. In case of a failure of a memory reading via the `m_r` function, this function will fail in the skeletal semantics interpretation monad. Then, the following operators use these to encapsulate the result in the new PCF monad. The evaluation of the new constructors is straightforward. Note that the evaluation of pure terms (such as `Add`) does not need to be modified: defining the correct `bind` is sufficient to upgrade the semantics to the new monad.

If we compare to related frameworks—namely K [63], Ott [66], and Lem [55]-, there is no evidence that these languages can easily access the continuation of a program, hence introducing new features, such as the state, would mean cluttering the code with operations for handling them. Otherwise, if we consider the work of Wadler [72], we cannot reproduce the approach as Skel does support modules and functors. Nevertheless, the purpose of the languages we are writing is a substantial difference between the two works. Our approach shows how to write formalizations in a tiny framework, not interpreters in some complex meta-language such as OCaml or Haskell.

An affinity exists with the modular SOS [46], but we underline a substantial difference: we write machine code while modular SOS is a paper technique.

Instantiation

Again, to generate a working interpreter, after a run of `necroml`, one must only extend the `Types` functor with a definition to the location type, and the state, which is a map from locations to values. The state type is defined similarly to the environment, relying on the OCaml Map module type. Then, one needs to provide an implementation of the three concrete operations for manipulating the state concretely.

3.4 A Fully Monadic Skeletal Semantics

We now show how to extend the Skel definition in the previous section by introducing the reader monad to handle environment effects—the goal is to have a fully monadic Skel definition of the language of Section 3.3-. We will alter the evaluation function to consider the monadic environment operations. However, this approach makes a formalization more solid to further extensions. Indeed, this approach is convenient while writing large-scale formalizations, showing how one can gradually introduce under-the-hood behaviors in a semantic description by only adding or changing a few lines of code.

We want to make the environment manipulation happen implicitly.

First, we define the reader monad type and redefine the `bind` and the `exccont` binders. We show the `bind` code in the following snippet.

```
type reader<a> := env → a
val bind<a,b> (v:reader<st<exc<a>>>)(f:(a → reader<st<exc<b>>>)) :
              reader<st<exc<b>>> =
  λs:env -> λm:state →
    let (v',m') = v s m in
    branch let Exc = v' in let r = throw<b> () in r s m'
    or     let Ok v'' = v' in f v'' s m'
    end
binder @ := bind
```

The reader monad, also called *environment monad*, serves to write programs in which all the operations may **read** data from an environment. In the `bind` shown above, we decided to combine the state, the exception, and the reader monad to obtain the following signature: ENV $\rightarrow$ STATE $\rightarrow$ (EXC a, STATE), which in Skel is written as `reader<st<exc<a>>>`. Similarly to the `bind`, we write the `exccont`.

Then, as the reader monad is used to read an environment to provide data, we define a function `ask` on the computation of type `reader<st<exc<env>>>`. This function provides an environment to a program.

```
val ask : reader<st<exc<env>>> = λs:env → λm:state ->  ((Ok<env> s),m)
```

We can provide, to the continuations, manipulations of the environment via the usual `local` function. We write this function's signature similarly to the `bind`'s and the `exccont`'s ones.

```
val local<a> (s:reader<st<exc<env>>>) (f: () → reader<st<exc<a>>>) :
              reader<st<exc<a>>> =
  λs_prev:env →
  λm:state → let (Ok s_new,m') = s s_prev m in
             f () s_new m'
```

Accordingly, we update the `output` alias type to be the one of the combination with the reader monad, and all the functions with the output sort of `type output`. Figure 3.11 shows the four monadic operations: `return` that returns a value in the `output` type, `newref` for a new allocation in the state, and `envGet` and `envExt`, the equivalent of the `getEnv` and `extEnv` functions, but returning a computation.

One of the purposes of updating the semantics is to manipulate the environment implicitly. Hence, we also define two functions for producing closures, either "normal" closures (`makeClosure`) or recursive ones (`makeRecursiveClosure`), and a function to get the environment from a closure (`closureEnv`). We show these three functions in Figure 3.12.

Finally, we update the evaluation function semantics. Let's start from its signature, which now changes to `val eval (e:expr) : output`. The reader can notice that we do not explicitly mention the environment (see Figure 3.10 for the old signature). Then, regarding the body of the `eval` function, we update only the constructors code manip-

```
type output := reader<st<exc<value>>>
val return<a> (v:a) : reader<st<exc<a>>> =
  λ_:env -> λm:state -> (Ok<a> v,m)

val newref (v:value) : output =
  λ_:env ->
  λm:state -> let ls = alloc v in
              let (l,m') = ls m in
              (Ok<value> (Loc l),m')

val envGet (x:var) : output =
    let s =@ ask in
    let v = getEnv s x in ret v
type env_output := reader<st<exc<env>>>
val envExt ((x,v):(var,value)) : env_output =
    let s =@ ask in
    let s' = extEnv s x v in
    return<env> s'
```

Figure 3.11: Sample of the updated functions

ulating the environment, which are the variable term evaluation, the λ-term definition, the application, the let-in construct, and the fixed-point operator.

Let us start with the variable assignment `Var`. The semantics shown in Figure 3.8 is using the `getEnv` function to get the mapping of a variable `x` to a value `v` in an environment `s`, if any. Then return the value as a computation by applying the `ret` function to `v`. With the reader monad, we can return the result directly by applying the monadic function `envGet` to x, returning a value encapsulated in the monadic type.

```
let Var x = e in envGet x
```

Then, we modify the λ-term's `eval` semantics. Evaluating $\lambda x.eb$ means to create a closure for `x,eb,s`, where `s` is the current environment. As the environment is no more explicit, we must ask for it, and only then create the closure. We designed a function to deal with this before, `makeClosure`. The constructor evaluation simply results in the following snippet.

```
val makeClosure ((x,b):(var,expr)) : output =
    let s =@ ask in
    ret (Clos (x,b,s))
val makeRecursiveClosure ((x,b,f):(var,expr,var)) : output =
    let s =@ ask in
    ret (ClosRec (x,b,s,f))
val closureEnv (v:value) : env_output =
    branch let Clos (_,_,s) = v in return<env> s
    or     let ClosRec (_,_,s,_) = v in return<env> s
    or     let Nat _ = v in throw<env>()
    or     let Unit = v in throw<env>()
    or     let Loc _ = v in throw<env>()
    end
```

Figure 3.12: Functions for creating closures. The output is a computation in the monad.

```
let Lam (x,eb) = e in makeClosure(x,eb)
```

Regarding the application `e1 e2`, in Figure 3.8, we first evaluated `e1` in the current environment `s`, then evaluated `e2` in `s` and extended the environment accordingly on the resulting closure whether `e1` is recursive or not. In the fully monadic semantics, we do the same, feeding the new current environment to the continuation via the local `local` monadic binder, and using `closureEnv` and `envExt` to "automate" respectively reading an environment from a closure and extending or updating it. We show the code below.

```
let App (e1,e2) = e in
let cl =@ eval e1 in
let w =@ eval e2 in
branch let Clos (x,eb,_) = cl in
       closureEnv cl;%local
       envExt(x, w);%local
       eval eb
or     let ClosRec (x,eb,_,f) = cl in
       closureEnv cl;%local
       envExt(x, w);%local
       envExt(f, cl);%local
       eval eb
end
```

For the let-in construct, in the PCF semantics, first, we evaluated the expression to a value `v` to be mapped to a variable name `x` in an environment `s`, then extended the environment `s` with the mapping $x \to v$, calling it `s'`, and finally returned the evaluation of the continuation in `s'`. The new code results in the following listing, where the `eval` call does not have the environment parameter anymore, and the computation contains the new environment, resulting from the application `envExt (x, v)`, which is directly bound to the continuation `eval eb`.

```
let LetIn (x,ex,ec) = e in
let v =@ eval ex in
envExt(x, v);%local
eval ec
```

Finally, similarly to what we did with the application case, modifying the fixed-point `eval` code means simply using the `makeRecursiveClosure`. We show this below.

```
let Fix (f, x, ex) = e in
makeRecursiveClosure(x,ex,f)
```

The rest of the evaluation function stays the same to what depicted in Figure 3.8 and 3.10. Concretely, the code evaluating the addition stays the same, as the rest of

the operations. One has only to delete the environment parameter from the `eval` calls.

```
let Add (e1,e2) = e in
let Nat n1 =@ eval e1 in
let Nat n2 =@ eval e2 in
let n = add(n1,n2) in ret (Nat n)
```

Introducing a posteriori the reader monad needed some effort to define some new monadic helper functions and rewrite the parts of the semantics involving the environment. Nevertheless, the semantics presented will not change in the next section, showing that the approach is highly maintainable. The only change in writing the language extension of the next section is how we will deal with the monads, as we introduce one more new effect. The two language semantics presented in Section 3.3 are behaviourally equivalent. We did not prove this property formally.

Instantiation

Introducing the reader monad in the semantic definition does not impact the instantiation of the interpreter. Indeed, we did not introduce any other new unspecified type or term. Hence, a user must only run `necroml` again and then use the same instantiation file of the stateful PCF.

3.5 Explicit Contination Manipulation

This section presents our most involved example by adding *delimited computations* to the language, following [31]. This complex transfer of control flow lets one start a computation, and this computation can suspend itself (*yield*) to return control to the calling computation, which can later resume the suspended computation. We combine this extension with exceptions and stateful computations. To this end, we use a *continuation monad* described in Section 3.5.2. To present the expected semantics, we describe it in the form of an abstract machine. Before these technical details, we give an intuition of the desired semantics through a series of examples in pseudo-code. The concept presented in the section finds a concrete use-case as future additions to OCaml [67].

```
1  let x = {yield 1; yield 2; 3} in
2  handle x [
3    ret v1: v1,
4    cont v1 k1: let y = resume k1 in
5         handle y [
6           ret v2: v2,
7           cont v2 k2: let z = resume k2 in
8                handle z [
9                  ret v3: (100*v1 + 10*v2 + v3),
10                 cont v3 _: v3
11               ]]]
```

Figure 3.13: Delimited Sequence of Yield

In the latter, the authors define language constructs for handling non-local control flow by manipulating stacked continuations.

3.5.1 Program Examples

Our first example is described in Figure 3.13. In Line 1, we assign the evaluation of the delimited computation `{yield 1; yield 2; 3}` to `x` (delimited computations are enclosed by curly braces). In this evaluation, `yield 1` suspends the computation and it returns both the value `1` and the continuation `{yield 2; 3}`. An *output* is thus a finished computation (with an actual value), or a suspension (with a value and a continuation). The `handle` operator takes such an output and two expressions, one to be executed if the computation is finished (field `ret`), and one if it is suspended (field `cont`). This can be seen in line 2 to 4. As `x` holds a suspension, the `cont` field is selected. It binds the value `1` to `v1` and the continuation to `k1`. This continuation is then resumed with the `resume` operator. Control is now passed to the continuation `k1` which corresponds to the code `{yield 2; 3}`. Evaluating this returns once again a suspended computation with value `2` and continuation `{3}`, which is bound to `y`. This suspended computation is handled one line 5, and the code on line 7 is then called. The continuation is once again resumed, and this time the delimited computation terminates with value `3`. Handling this output is done on line 8, and as it is finished, the `ret` case is selected. The expression `(100*v1 + 10*v2 + v3)` is then computed, returning `123`.

An intuitive way to consider such computations is to imagine them as a stack, where

```
1  let x ={let y = 1 in let z = yield y in (100*y + 10*z + yield 0)} in
2  handle x [
3    ret v1: v1,
4    cont v1 k1: let y = resume (k1, (v1 + 1)) in
5         handle y [
6           ret v2: v2,
7           cont v2 k2: let z = resume (k2, 3) in
8                handle z [
9                  ret v3: v3;
10                 cont v3 _: v3
11               ]]]
```

Figure 3.14: Two Ways Communication through Yield

the top of the stack is currently being executed. Creating a delimited computation `{e}` adds it to the stack and evaluates it. Since `e` may itself start a delimited computation, the stack can be arbitrarily tall. Once a computation returns, either suspended because of `yield` or finished, the next computation in the stack starts evaluating. If the stack is empty, then the program ends.

This example shows that the delimited section may communicate values to its context (the next computation in the stack) using `yield`. We can make this communication bidirectional by also passing a value when resuming a continuation. We show it in Figure 3.14. Now each time we resume a continuation, we send back a resumption value. Indeed, the first yield in the example returns the value 1, which is the value the variable `y` is bound to, and the rest of the delimited computation as continuation. When we resume, in line 4, we resume the continuation `k1`, with a resumption value `(v1 + 1)`, with `v1` bound to 1. The expression `let z = yield y in` on line 1 creates the binding $[z \to 2]$. The delimited section suspends again at the evaluation of `yield 0`. Line 7 resumes the computation communicating back the value 3. Finally, the delimited section finishes computing returning to the handler the value 123, returned as final value of the program.

3.5.2 Syntax and Semantics

To formally capture this semantics, we need to explicitly manipulate stacks of continuations. We thus extend the syntax of the language presented in Section 3.4.

Formal Syntax

$$
\begin{aligned}
&\cdots \\
\textsc{Cout}\ o &: \textsc{St}(\textsc{Exc}(\textsc{Val})) \\
\textsc{Cont}(\alpha)\ \kappa &: \alpha \to \textsc{Cstack}(\alpha) \to \alpha \\
\textsc{Cstack}(\alpha)\ \pi &::= \epsilon \mid \kappa_\alpha :: \pi_\alpha \\
\textsc{Value}\ v &::= \cdots \mid (v, \kappa_o) \mid \kappa_o \\
\textsc{ContM}(\alpha)\ \rho &: \textsc{Cont}(\alpha) \to \textsc{Cstack}(\alpha) \to \alpha \\
\textsc{Expr}\ e &::= \cdots \mid \{e\} \mid \texttt{yield}\ e \mid \texttt{handle}\ e_h\ \ [\texttt{ret}\ x_{vr} : e_r\,, \texttt{cont}\ x_{vc}\ x_k : e_c] \\
&\qquad\quad \mid \texttt{resume}(e_\kappa, e_v)
\end{aligned}
$$

```
...
type out := st<exc<value>>
type cont<a> := a → cstack<a> → a
type cstack<a> = | Nil | Cons (cont<a>, cstack<a>)
type value = | ... | Susp (value, cont<out>) | Cont cont<out>
type contM<a> := cont<a> → cstack<a> → a
type expr = | ... | Run expr | Yield expr
| Handle (expr, (var,expr), (var,var,expr))
| Resume (expr, expr)
```

ContM(α) is the monadic type constructor for the continuation monad. Given ρ of type ContM(α), κ of type Cont(α) and π of type Cstack(α), $\rho\ \kappa\ \pi$ produces a value of type α, the result type of the program.

The Cont(α) type represents continuations: it is a function that expects a value and a stack of continuations to return a value. The type Cstack(α) represents a stack of continuations. The stack may be empty, denoted with ϵ, or containing a continuation and the rest of the stack, represented as $\kappa :: \pi$.

We extend also the Value type with suspension depicted as a tuple (v, κ), with a value v and a continuation κ. We also add to values continuation themselves. For instance, the evaluation of the sequence `(yield e1; e2)` returns a suspension value $(v1, k2)$, where $v1$ is the result of evaluating `e1`, and $k2$ corresponds to the suspended

evaluation of e2.

$$\langle x, \kappa, \pi, \mu, \sigma\rangle_e \to [\kappa, \pi, \mu, \sigma, Ok\ v]_c \qquad \text{with } v = \sigma(x) \tag{3.1}$$
$$\langle e_1 + e_2, \kappa, \pi, \mu, \sigma\rangle_e \to \langle e_1, (\square + e_2) :: \kappa, \pi, \mu, \sigma\rangle_e \tag{3.2}$$
$$\langle n, \kappa, \pi, \mu, \sigma\rangle_e \to [\kappa, \pi, \mu, \sigma, Ok\ n]_c \tag{3.3}$$
$$[(\square + e_2) :: \kappa, \pi, \mu, \sigma, Ok\ n_1]_c \to \langle e_2, (n_1 + \square) :: \kappa, \pi, \mu, \sigma\rangle_e \tag{3.4}$$
$$[(n_1 + \square) :: \kappa, \pi, \mu, \sigma, Ok\ n_2]_c \to [\kappa, \pi, \mu, \sigma, Ok\ n]_c \qquad \text{with } n = n_1 +_{\mathbb{Z}} n_2 \tag{3.5}$$
$$[f :: \kappa, \pi, \mu, \sigma, Exc]_c \to [\kappa, \pi, \mu, \sigma, Exc]_c \qquad \forall e_2, f \neq \texttt{try}\{\square\}\{e_2\} \tag{3.6}$$
$$[\texttt{try}\{\square\}\{e_2\} :: \kappa, \pi, \mu, \sigma, Exc]_c \to \langle e_2, \kappa, \pi, \mu, \sigma\rangle_e \tag{3.7}$$
$$[\texttt{try}\{\square\}\{e_2\} :: \kappa, \pi, \mu, \sigma, Ok\ v]_c \to [\kappa, \pi, \mu, \sigma, Ok\ v]_c \tag{3.8}$$
$$\langle e_{cl} e_v, \kappa, \pi, \mu, \sigma\rangle_e \to \langle e_{cl}, \square e_v :: \kappa, \pi, \mu, \sigma\rangle_e \tag{3.9}$$
$$[\square e_v :: \kappa, \pi, \mu, \sigma, Ok\ (x, e_x, \sigma_{cl})]_c \to \langle e_v, (x, e_x, \sigma_{cl})\square :: \kappa, \pi, \mu, \sigma\rangle_e \tag{3.10}$$
$$[(x, e_x, \sigma_{cl})\square :: \kappa, \pi, \mu, \sigma, Ok\ v]_c \to \langle e_x, \kappa, \pi, \mu, (\sigma_{cl} + x \to v)\rangle_e \tag{3.11}$$
$$[\square e_v :: \kappa, \pi, \mu, \sigma, Ok\ (x, e_x, \sigma_{cl}, f)]_c \to \langle e_v, (x, e_x, \sigma_{cl}, f)\square :: \kappa, \pi, \mu, \sigma\rangle_e \tag{3.12}$$
$$[(x, e_x, \sigma_{cl}, f)\square :: \kappa, \pi, \mu, \sigma, Ok\ v]_c \to \langle e_x, \kappa, \pi, \mu, (\sigma_{cl} + ((x \to v) + (f \to (x, e_x, \sigma_{cl}, f))))\rangle_e \tag{3.13}$$
$$\langle e_l := e_v, \kappa, \pi, \mu, \sigma\rangle_e \to \langle e_l, \square := e_v :: \kappa, \pi, \mu, \sigma\rangle_e \tag{3.14}$$
$$[\square := e_v :: \kappa, \pi, \mu, \sigma, Ok\ l]_c \to \langle e_v, l := \square :: \kappa, \pi, \mu, \sigma\rangle_e \tag{3.15}$$
$$[l := \square :: \kappa, \pi, \mu, \sigma, Ok\ v]_c \to [\kappa, \pi, \mu', \sigma, Ok\ ()]_c \qquad \mu(l, v) = \mu' \tag{3.16}$$
$$\langle \{e\}, \kappa, \pi, \mu, \sigma\rangle_e \to \langle e, \texttt{id}, \kappa :: \pi, \mu, \sigma\rangle_e \tag{3.17}$$
$$[\texttt{id}, \kappa :: \pi, \mu, \sigma, r]_c \to [\kappa, \pi, \mu, \sigma, r]_c \tag{3.18}$$
$$\langle \texttt{yield}\ e, \kappa, \pi, \mu, \sigma\rangle_e \to \langle e, \texttt{yield}\ \square :: \kappa, \pi, \mu, \sigma\rangle_e \tag{3.19}$$
$$[\texttt{yield}\ \square :: \kappa, \kappa' :: \pi, \mu, \sigma, Ok\ v]_c \to [\kappa', \pi, \mu, \sigma, Ok\ (v, \kappa)]_c \tag{3.20}$$
$$\langle \texttt{handle}\ e_h\ \ [\texttt{ret}\ x_{vr} : e_r\,, \texttt{cont}\ x_{vc}\ x_k : e_c], \kappa, \pi, \mu, \sigma\rangle_e \to \langle e_h, \texttt{handle}\ \square\ \ [\texttt{ret}\ x_{vr} : e_r\,, \texttt{cont}\ x_{vc}\ x_k : e_c] :: k, \pi, \mu, \sigma\rangle_e \tag{3.21}$$
$$[\texttt{handle}\ \square\ \ [\texttt{ret}\ x_{vr} : e_r\,, \texttt{cont}\ x_{vc}\ x_k : e_c] :: k, \pi, \mu, \sigma, Ok\ v]_c \to \langle e_r, \texttt{handle}\ v :: \kappa, \pi, \mu, (\sigma + x_{vr} \to v)\rangle_e \qquad v \text{ pure} \tag{3.22}$$
$$[\texttt{handle}\ \square\ \ [\texttt{ret}\ x_{vr} : e_r\,, \texttt{cont}\ x_{vc}\ x_k : e_c] :: \kappa, \pi, \mu, \sigma, Ok\ (v, k)]_c \to \langle e_c, \texttt{handle}\ v :: \kappa, \pi, \mu, (\sigma + x_{vc} \to v + x_k \to k)\rangle_e \tag{3.23}$$
$$\langle \texttt{resume}(e_k, e_v), \kappa, \pi, \mu, \sigma\rangle_e \to \langle e_k, \texttt{resume}(\square, e_v) :: \kappa, \pi, \mu, \sigma\rangle_e \tag{3.24}$$
$$[\texttt{resume}(\square, e_v) :: \kappa, \pi, \mu, \sigma, Ok\ \kappa']_c \to \langle e_v, \texttt{resume}(\kappa', \square) :: \kappa, \pi, \mu, \sigma\rangle_e \tag{3.25}$$
$$[\texttt{resume}(\kappa', \square) :: \kappa, \pi, \mu, \sigma, Ok\ v]_c \to [\kappa', \kappa :: \pi, \mu, \sigma, Ok\ v]_c \tag{3.26}$$

Figure 3.15: Abstract Machine for PCF + state + delimited continuations.

We extend the expressions with the following language constructors:

- `yield e`, for suspending a computation while returning the evaluation of `e`;
- `{e}`, for starting a delimited computation;
- `handle eh[ret xvr: er, cont xvc xk: ec]`, to inspect the result of the evaluation of `e`, distinguishing between a normal return or a suspension;
- `resume (ek,ev)`, where `ek` evaluates to a continuation, to resume it with the resumption value computed from `ev`.

To give a formal definition of our language, we need to switch paradigms, introduced in Section 1.3, as big step semantics are not convenient to describe suspended computations. To this end, one typically uses context-based reduction semantics or abstract machines. We choose the latter as the continuations are made explicit in that model. Our abstract machine is defined in Figure 3.15, with some similar rules omitted.

The abstract machine may be in one of two modes, one for evaluating expressions, and one for applying continuations. The evaluation mode, written as $\langle e, \kappa, \pi, \mu, \sigma \rangle_e$, evaluates e with a continuation κ, a continuation stack π, a state μ and an environment σ. Similarly, the continuation mode, denoted by $[\kappa, \pi, \mu, \sigma, r]_c$ applies the continuation κ to the computed result r. A result is either an exception Exc or a normal result $Ok\ v$, where v is either a pure value as before or a suspension (v, κ).

A continuation is a stack of *frames* f that contain an expression with a hole. For instance, when computing the first argument of an addition $t_1 + t_2$, the frame is $\square + t_2$ (rule 3.2). When evaluating a constant, the machine switches to continuation mode (rule 3.3). Applying the frame $\square + t_2$ to a value n_1 switches back to evaluation mode for t_2 with frame $n_1 + \square$ (rule 3.4). Finally, applying this frame to a value n_2 computes $n_1 + n_2$ and passes it to the next frame by staying in continuation mode (rule 3.5).

Rules 3.6 to 3.8 show how exceptions are handled: if an exception is raised, it is caught by the first `try` frame. Otherwise, applying a `try` frame to a value simply passes control to the rest of the continuation.

Rules 3.9 to 3.13 deal with function and recursive function application. Rules 3.14 to 3.16 show how the heap may be mutated.

We now turn to two programs that involve delimited computations: `{1+2} + 3` and `{(yield 1)+2}`. The initial configuration of the abstract state machine for the first program is $\langle$`{1+2} + 3`$, \mathtt{id}, \epsilon, \mu, \sigma \rangle_e$, where `id` represents the empty continuation. Applying

rule 3.2 results in $\langle \texttt{\{1 + 2\}}, (\square \texttt{ + 3}) :: \texttt{id}, \epsilon, \mu, \sigma \rangle_e$. Next, the machine uses rule 3.17 to enter the delimited computation. It push the current continuation on the continuation stack and starts evaluation `1+2` with a new empty continuation, resulting in the configuration $\langle \texttt{1+2}, \texttt{id}, ((\square \texttt{ + 3}) :: \texttt{id}) :: \epsilon, \mu, \sigma \rangle_e$. The addition is then evaluated as above, before reaching configuration $[\texttt{id}, ((\square \texttt{ + 3}) :: \texttt{id}) :: \epsilon, \mu, \sigma, Ok\ 3]_c$. The delimited computation is finished as only the empty continuation remains, so the previous continuation is restored with the computed value (rule 3.18): $[(\square \texttt{ + 3}), \texttt{id}, \epsilon, \mu, \sigma, Ok\ 3]_c$. The computation then proceeds to the final state $[\texttt{id}, \epsilon, \mu, \sigma, Ok\ 6]_c$.

The second program starts with configuration $\langle \texttt{\{(yield 1)+2\}}, \texttt{id}, \epsilon, \mu, \sigma \rangle_e$. The derivation proceeds in a similar way as the previous example till we reach the `yield` with a configuration $\langle \texttt{yield 1}, (\square\texttt{+2}) :: \texttt{id}, \texttt{id} :: \epsilon, \mu, \sigma \rangle_e$. Now by applying rule 3.19, we first evaluate the constant expression `1` in the configuration $\langle 1, \texttt{yield } \square :: (\square\texttt{+2}) :: \texttt{id}, \texttt{id} :: \epsilon, \mu, \sigma \rangle_e$. Hence, we end up in the continuation configuration $[\texttt{yield } \square :: (\square\texttt{+2}) :: \texttt{id}, \texttt{id} :: \epsilon, \mu, \sigma, Ok\ 1]_c$. Rule 3.20 restores the continuation at the top of the stack as current continuation (here `id`), and a suspension as result:
$[\texttt{id}, \epsilon, \mu, \sigma, Ok\ (1, (\square\texttt{+2}) :: \texttt{id})]_c$. As the suspension is not resumed, the computation ends here and the addition is not performed.

We next turn to two examples using the `handle` construct.

```
(* A *)
handle ({1 + 2} + 3) [
  ret v1: v1,
  cont _ _: skip
]
```

```
(* B *)
handle {yield 1 + 2} [
  ret v1: v1,
  cont v1 k1: let x = resume (k1,v1 + 3) in
       handle x [
         ret v2: v2,
         cont v2 _: v2 ]]
```

Consider the following as initial configuration for listing A.

$$\langle \texttt{handle ({1 + 2} + 3) [ret v1:v1,cont v1 _:v1]}, \texttt{id}, \epsilon, \mu, \sigma \rangle_e$$

By applying rule 3.21, the expression `{1 + 2} + 3` is first evaluated, resulting in a continuation configuration of the form:
$[\texttt{handle } \square \texttt{ [ret v1:v1,cont _ _:skip]} :: \texttt{id}, \epsilon, \mu, \sigma, Ok\ 6]_c$. This time the evaluation is not yet complete, as the current continuation holds the remainder of the `handle` construct. The value in the continuation configuration is 6, a pure value, so the next

thing to do is to evaluate the expression in the `ret` field of the handler. By rule 3.22, this result in $\langle \texttt{v1}, \texttt{id}, \epsilon, \mu, (\sigma + \texttt{v1} \rightarrow 6) \rangle_e$. The next step accesses the environment—rule 3.1-to lead to a final configuration $[\texttt{id}, \epsilon, \mu, \sigma, Ok\ 6]_c$.

We now turn to listing B. After evaluating the `yield` construct, we reach the following configuration.

$$[\texttt{handle}\ \square\ \texttt{[ret v1:v1,cont v1 k1:...]} :: \texttt{id}, \epsilon, \mu, \sigma, Ok\ (1, (\square + 2) :: \texttt{id})]_c$$

The evaluation of the expression `{yield 1 + 2}` returns a suspension which is a pair with a value 1 and the reminder of the delimited section computation $(\square + 2) :: \texttt{id}$. As the suspension is not a pure value, the evaluation proceeds by applying rule 3.23 on the latter configuration. We obtain the following, where σ' is $\sigma + (\texttt{v1} \rightarrow 1) + (\texttt{k1} \rightarrow (\square + 2) :: \texttt{id})$.

$$\langle \texttt{let x = resume(k1,v1 + 3) in ...}, \texttt{id}, \epsilon, \mu, \sigma' \rangle_e$$

We first apply the evaluation rule for let (not shown), yielding this configuration.

$$\langle \texttt{resume(k1,v1 + 3)}, (\texttt{let x = } \square \texttt{ in ...}) :: \texttt{id}, \epsilon, \mu, \sigma' \rangle_e$$

We apply rule 3.24 to start the evaluation of `k1`, then rule 3.25 to start the evaluation of `v1 + 3`. We then reach the following configuration, the first square of the `resume` is part of the stored continuation `k1`, whereas the second square is waiting for the resumption value.

$$[\texttt{resume}((\square + 2) :: \texttt{id}, \square) :: (\texttt{let x = } \square \texttt{ in ...}) :: \texttt{id}, \epsilon, \mu, \sigma', Ok\ 4]_c$$

Resuming a continuation means evaluating in, passing to it the last value computed. The current continuation $((\texttt{let x = } \square \texttt{ in ...}) :: \texttt{id})$ is itself stored at the top of the stack. After applying rule 3.26, we obtain the following configuration.

$$[(\square + 2) :: \texttt{id}, ((\texttt{let x = } \square \texttt{ in ...}) :: \texttt{id}) :: \epsilon, \mu, \sigma, Ok\ 4]_c$$

After evaluating the addition and applying rule 3.18 for the identity continuation, we

```
type output := reader<contM<st<exc<value>>>>>
val eval (e:expr) : output =
 branch ...
 or     let Add (e1,e2) = e in
        let Nat n1 =@ eval e1 in
        let Nat n2 =@ eval e2 in
        let n = add(n1,n2) in ret (Nat n)
 or     let Run e' = e in ();%pushCont
        eval e'
 or     let Yield e' = e in
        let v =@ eval e' in susp v
 or     let Handle (eh,(xvr,er),(xvc,xk,ec)) = e in
        let v =@ eval eh in
        branch let v = getPure v in
               envExt(xvr, v);%local eval er
        or     let (v,k) = getSusp v in
               envExt(xvc, v);%local envExt(xk, (Cont k));%local
               eval ec
        end
 or     let Resume (ek,ev) = e in
        let kv =@ eval ek in
        let v =@ eval ev in
        let k = getCont kv in resume k v
end
```

Figure 3.16: Yield Language in Skel

obtain the following configuration.

$$[\texttt{let x = } \square \texttt{ in handle x [ret v2: v2, cont v2 \_: v2]}, \texttt{id}, \epsilon, \mu, \sigma, Ok\ 6]_c$$

The rest of the derivations follows the pattern of the first `handle` example, leading to a final configuration $[\texttt{id}, \epsilon, \mu, \sigma, Ok\ 6]_c$.

3.5.3 A Stateful PCF Language with Yield and Exceptions in Skel

We now present the Skel formalization of the language described in Section 3.5.2. As before, we change the meaning of monadic binders and returns, to keep the main

description of the semantics unchanged.

Our semantics is described in Figure 3.16. We highlight the addition rule, showing it is not modified. As before, the behavioral complexity of dealing with continuations for previously existing constructs is delegated to the monadic binders `binder @ := bind` and `binder ? := exccont`.

The `bind` function works as follows. It expects a computation `w` in the monad `reader<contM<st<exc<a>>>>`, a function `f` to apply after the computation, and it should return a term in the same monad. To do so, we first request (using λ abstractions) the environment `s`, the continuation `k` to apply after `w` and `f`, the current continuation stack `ks`, and the current state `m`. We then execute `w` with a new continuation, detailed below, the stack `ks`, and the state `m`. The intuition behind the new continuation is that it should proceed with `f` if the result is not an exception, passing `k` to it as continuation, otherwise it should return the exception using `k`. The new continuation is defined as follows. It first take a computation `vse` in the state and exception monad, the current stack `ks1`, and the current state `m1`. It then applies `vse` to `m1` to obtain an exception value `ve` and a new state `m2`. The exception value is then inspected. If it is an exception, it is passed (embedded in the state monad) to the continuation `k` with stack `ks1` and updated state `s2`. Otherwise, it is a value `v` which we pass to `f`, with the environment `s`, continuation `k`, stack `ks1`, and state `m2`.

The behavior of `extcont` is similar, reversing the two inner branches. Note that we could simplify the code for the binders by removing some eta-expansions for state `m` and stack `ks`, we keep them for clarity. We show the code of the `bind` function below.

```
val bind<a> (w:reader<contM<st<exc<a>>>>)
            (f:a → reader<contM<st<exc<a>>>>) :
             reader<contM<st<exc<a>>>> =
  λs:env →
  λk:cont<st<exc<a>>> →
  λks:cstack<st<exc<a>>> →
  λm:state → w s (λvse:st<exc<a>> →
                  λks1: cstack<st<exc<a>>> →
                  λm1:state →
                    let (ve,m2) = vse m1 in
                    branch let Exc = ve in
                           k (λm3:state → (Exc<a>,m3)) ks1 m2
                    or     let Ok v = ve in
                           let w' = f v in w' s k ks1 m2
                    end) ks m
```

The rule for a new delimited computation `{e}` is defined in the `Run` branch. It pushes the current continuation on the continuation stack and runs `e` using the identity continuation (Similar to rule 3.17). To simplify the code, we use the monadic binder `pushCont` to push the continuation. We show the code below.

```
val id (v:out)(ks:cstack<out>):out =
 λm:state →
  branch let Nil = ks in v m
  or     let Cons (k, ks') = ks in k v ks' m
  end

val pushCont (_:())(f:()→output):output =
 λ_:env → λk:cont<out> → λks:cstack<out> →
   f () id (Cons<out>(k,ks))
```

This snippet also contains the code for the identity continuation, which returns the final value if the stack is empty, otherwise it passes this value to the next continuation on the stack, as in rule 3.18 of Figure 3.15.

The `Yield` rule is straightforward with our monad. We first evaluate its argument expression to `v`, then we call `susp` to suspend the current computation `k`. This function extracts the continuation `k'` at the top of the stack and calls it with the value `Susp(v,k)`. We present its definition in the following listing.

```
val susp (v:value):output =
 λ_:env ->
 λk:cont<out> →
 λks:cstack<out> →
  let Cons(k',ks') = ks in
  k' (λs:state →(Ok<value>(Susp (v,k)),s)) ks'
```

The `Handle` construct defines two cases, depending on the result of the evaluation of `eh`. In the pure case `v`, the environment `s` is extended with a binding `xvr` $\rightarrow$ `v` and we evaluate `er`. This case corresponds to the abstract machine rule 3.22. In the suspension case `(v,k)`, the environment is extended by `xvc` $\rightarrow$ `v` and `xk` $\rightarrow$ `k`, then `er` is evaluated. This follows rule 3.23.

Finally, `Resume(ek,ev)` is the constructor for resuming a suspended computation. We first evaluate `ek` to a continuation `k` (rule 3.24) and `ev` to a value `v` (rule 3.25). Then, the `resume k v` call performs the behavior of rule 3.26. Basically, it puts `k` as current computation to evaluate, and pushes the actual current continuation on top of the stack. The continuation `k` is resumed with the value `v` and the new stack. We show the code below.

```
val resume (kv:cont<out>)(v:value):output =
 λ_:env ->
 λk:cont<out> →
 λks:cstack<out> →
  kv (λm:state →(Ok<value> v,m))
     (Cons<out> (k,ks))
```

In Figure 3.17, we show a final example of program. In this example, we interleave state updates in a delimited computation and in the calling code. This program shows that the state is global. Communication between delimited code is bidirectional, both in an explicit way, through suspensions and resumption values, and in an implicit way

```
let x = { let y = &1 in
              let x1 = !y in
              let x2 = !(yield y) in
              yield(y := 1);
              (100*x1 + 10*x2 + (!y)) } in
handle x [
       ret v1: v1,
       cont r k1:
              r := 2;
              handle (resume (k1,r)) [
                     ret v2: v2,
                     cont _ k2:
                            r := !r + 2;
                            handle (resume (k2,r)) [
                                   ret v3: v3,
                                   cont v3 _: v3]]]
```

Figure 3.17: Ping-Pong State Updates through Yield

through state updates. The result of the program is again 123.

Instantiation

Once the user generate via `necroml` the file `.ml`, it can just use the same instantiation file defined for the stateful PCF (Section 3.3).

3.6 Related Work

There are many criteria to evaluate the formalization of a programming language: it should be easy to write and to reuse, it should be close to the specification if there is one, it should be executable to be tested, and it should be usable, for instance to prove properties of programs or the language itself, or to mechanically manipulate the semantics. In this Chapter, we do not directly address the last two points, but we remark that skeletal semantics can be translated to OCaml interpreters using [42], and to Coq formalizations using [50]. Recent work uses this approach to derive a certified

interpreter [1].

The formalization of a language may take several forms: it can be given as an interpreter, it can be defined in a proof assistant, or it can be described in a framework providing a meta-language and tools to manipulate the formalization.

An interpreter may be considered to be a formalization of a language if it is written in a clear way and if the host language has itself a clear semantics. Consider for instance engine262 [22]: the code is very close to the specification of JavaScript, but the host language is JavaScript itself, which is quite complex. In fact, engine262 uses JavaScript's generators to formalise them,[2] hence it does not clarify how they work. Implementations as specifications can be useful to give an intuition of the semantics of a language, but they do not provide a simple way to manipulate the semantics or to prove properties.

Many languages have been directly formalised in Coq [70], such as C in the setting of CompCert [39] or JavaScript [6], while others have been formalised in Isabelle/HOL [29], such as ML in the setting of CakeML [37]. Using extraction mechanisms, some of these semantics can be made executable. These works are impressive achievements, but they suffer from two shortcomings. First, replicating them for a new language requires a significant effort. Second, the semantics defined cannot be easily manipulated, for instance to mechanically transform them.

To alleviate these issues, one may use a framework in which the language is described, and from which implementations or formal definitions in proof assistants may be derived. These generated implementations may not be as elegant as when done by hand, but they can be easily adapted when needed, for instance switching from a big-step semantics to a small-step one, or going from a shallow embedding to a deep embedding. This is the approach followed by skeletal semantics, as well as many other frameworks such as Ott [66], Lem [55], and $\mathbb{K}$ [63]. The distinguishing feature of skeletal semantics is how simple the meta-language is: it does not come with any predefined semantics nor does it try to deal with complex syntactic features such as binders. The cost of this choice is that a skeletal semantics description of a language may require more work—complexity of the approach relies in monadic function definitions-, but the benefits is that skeletal semantics can be easily extended in the language itself, through the use of monads as illustrated in this Chapter. This is because skeletal pro-

2. `https://github.com/engine262/engine262/blob/206928332324f0ae95485aee9784dbf19e525b36/src/abstract-ops/generator-operations.mjs#L245`

grams are almost in Administrative Normal Form [64], thus removing the need to use a CPS transformation to have access to continuations [24]. It is not clear how one would add delimited computations to the previously cited frameworks. Another benefit, presented in Chapter 2, is the existence of a simple API to manipulate skeletal semantics, enabling the creation of many tools based on the same input language.

Chapter 4

A Faithful Description of ECMAScript in Skeletal Semantics

Contents

« Ora et labora »
- Saint Benedict

Introduction

This chapter shows that the Skeletal Semantics framework [8, 54] is suitable for mechanizing a real-world specification. To this end, we describe the ongoing formalization

of ECMAScript—formal semantics of JavaScript-algorithms in Skel. Differently from Chapter 3, the document presenting the language defines the evaluation as algorithms, allowing us to show that our approach is suitable also to mechanize this type of definition. We show that the resulting description is very close to the specification. In addition, we have formalized enough algorithms to be able to run simple JavaScript programs and thus test our approach.

The project is the first formalization of a real-world programming language in Skel, the meta-language used to write Skeletal Semantics. This work strongly supported the evolution of Skel, from the introduction of the notion of polymorphism to the addition of monads and first-class functions. This evolution is motivated by the need to have a formal semantics and a language expressive enough to capture the behavior of ECMAScript algorithms, ensuring a visual and a behavioral match between the formalization and the specification.

Our main effort is presenting the description of the mechanization of a core ECMAScript algorithm in Section 4.1, where we also present some crucial features of Skeletal Semantics, which significantly improve the readability of the mechanization, inspired by the approach presented in Chapter 3. In Section 4.2, we evaluate our approach, and finally, in Section 4.3, we discuss previous work on JS semantics.

Contribution

In this chapter we present:

- the design of a concise, readable, maintainable and textually close writing approach to model complex algorithmic specifications in Skel, namely ECMAScript.
- an ongoing formalization of ECMAScript in Skel.
- an instantiation of a JavaScript interpreter generated from a semantic framework.
- some executions of small JavaScript programs.

The contributions of this chapter are part of a national [34] and an international(PPDP'22) [33] publication.

4.1 ECMAScript Algorithms in Skel

In this section, we illustrate how we formalize ECMAScript(ES) algorithms in Skel. In Section 4.1.1, we provide some context about the ES specification. Then, in Sections 4.1.2, 4.1.3, and 4.1.4, we introduce methods to use the tiny Skel language for handling and combining ECMAScript's side-effects, ending up in a structured and systematic way for describing the algorithms. In Section 4.1.5, we present the formalization of the `GetValue`[1] *Internal Method* as representative of our approach. The `GetValue` method is one of the most referenced in ES, and is also enough complex to describe our systematic approach to formalizing of ES algorithms. Finally, in Section 4.1.6, we detail the current state of the work.

4.1.1 ECMAScript

ECMAScript is a large vernacular specification written in an imperative style. It is divided into 28 chapters and six appendices. Despite its complexity and verbosity, it provides a complete specification of the behavior of JS. However, some choices are left to the implementation, so a formalization must consider these unspecified pieces. We will explain later how we deal with them.

After an introductive part in chapters 1 to 5, where the notational and algorithmic conventions are defined, the document can be divided into three main functional groups.

Chapters 6, 7, and 9 give a taxonomy of ES *Data Types and Values*, providing each taxon with a definition of its operations and related invariant, followed by the definition of *abstract operations* (type conversion, comparison, object, and iterator operations), the *runtime environment*, and detailed classification of the type *Object* and its internal methods. This block of chapters gives a complete overview of the execution environment in which an ES program should be executed. Formalizing any language construct first requires a formalization of this execution environment.

Chapter 8 defines some syntax-directed operations. The central part of the specification, chapter 10 to chapter 16, describes the actual ES programming language. Chapters 10 and 11 focus on lexical units. Language constructs are given in chapters 12 to 15, where each construct is given with its syntactic specification and its eval-

1. `https://tc39.es/ecma262/2021/#sec-getvalue`

uation. These describe expressions, statements, functions, and scripts. Modules are defined in chapter 16.

Chapters 17 to 27 introduce the default components of the *Global Object*. In addition, a memory model is given in Chapter 28.

4.1.2 Challenges of the Formalization

The first step of the mechanization in the purely functional Skel language is to deal with the imperative nature of the specification, raising two issues.

First, there is a notion of an implementation-dependent state that can be mutated. More precisely, we define by *state* the aggregation of all the imperative data manipulated by the specification. We design it as a record that includes the Execution Context stack, a strictness boolean flag, and a pool of Maps holding *Execution Contexts*, *Environment Records*, *Realms*, *Script Records*, and *Objects*. In the Skel artefact, we have fully defined both the state and its helper functions. To remain close to the imperative specification, we design a Skel state monad in Figure 4.1. This monad lets us implicitly pass the state around, similarly to what we showed in Section 3.3. Note that we could have left these "implementation-dependent" elements of the specification unspecified, providing an implementation to this type and its function after generating an artefact.

Second, the specification often breaks the usual control flow by having a `return` in the middle of algorithms. We can capture such control flows by using nested branches (see below), but this significantly reduces the legibility of the mechanization. We thus define a *control flow* monad to simplify the mechanization. In addition, the specification itself introduces operators that behave much like an exception monad to deal with `break`, function returns, or exceptions.

We thus propose in Section 4.1.3 an exception monad that captures the behavior of ES monadic shorthands `?` and `!`,[2] and in Section 4.1.4, its extension to handle control-flow features. We claim the combination of the state monad with the control-flow and the exception ones dramatically simplifies our code, making the Skel description of ES algorithms easy to write and maintain. The result is close to the specification, as shown in Section 4.1.5.

2. `https://tc39.es/ecma262/2021/#sec-returnifabrupt-shorthands`

```
type state (* specified in the artefact *)
type st<a> := state → (a, state)

val st_bind<a,b> (v:st<a>) (f:(a → st<b>)) : st<b> =
    λs: state →
     let (v', s') = v s in f v' s'
val st_ret<a>: (v: a) : st<a> = λs: state → (v, s)
```

Figure 4.1: State Monad in Skel

Example of Implementation Dependent Choices: The State

The state is fully defined in the artefact, represented as a Skel record with eight fields. In the following paragraphs, we present a broad explanation of how we implemented data stored in the state. Most of the elements in the ES memory are left "implementation dependent", which gives much space for misinterpretation or different correct implementations. We tried to study the manuscript to describe these structures by retrieving information from the ES document.

Execution Contexts An execution context is an ES specification data structure used to keep track of the runtime evaluation of executable code by an ES implementation. In the document, there is a broad description of what concretely is this specification device. We quote its definition.

ECMAScript

> An execution context contains whatever implementation specific state is necessary to track the execution progress of its associated code.

This definition is complemented with a table providing a broad definition of how one can implement it. Indeed the table defines four fields:

- *Code evaluation state*: "Any state needed to perform, suspend, and resume evaluation of the code associated with this execution context".
- *Function*: "If this execution context is evaluating the code of a function object, then the value of this component is that function object. If the context is evaluating the code of a Script or Module, the value is null."

- *Realm*: “The Realm Record from which associated code accesses ECMAScript resources.”

- *ScriptOrModule*: “The Module Record or Script Record from which associated code originates. If there is no originating script or module, as is the case for the original execution context created in InitializeHostDefinedRealm, the value is null.”

Accordingly to the definition in the table, in Skel, we define a record defining eight fields, one for each bullet of the previous list plus three extra fields not defined in the table but accessed and modified by some algorithmic steps in the document.

```
type executionContext = (
     _CodeEvaluationState_: codeEvaluationState,
     _EC_Function_: maybeNull<loc_Object>,
     _Realm_ : option<loc_realmRecord>,
     _ScriptOrModule_ : maybeNull<scriptOrModule>,
     _VariableEnvironment_ : option<loc_EnvironmentRecord>,
     _LexicalEnvironment_:  option<loc_EnvironmentRecord>,
     _Generator_: maybeNull<loc_Object>
)
```

The type `maybeNull` is an option value similar to the classic `maybe` type in Haskell or the `option` type in OCaml. The difference is that, in this case, the options for `maybeNull` are between being `MN_Null`, referring to the ECMAScript value `null`, or being the polymorphic constructor `MN_NotNull` of type `a`, to say that the value is not `null`. Throughout the formalization, we define types similar to this to create an option between an ECMAScript pure value and different kinds of data. Another option is to reify by defining a sum type containing all the types in the formalization. This choice has been ignored as it is complicated to give a precise number of the types in the ES document as it is untyped. The type `executionContext` is extended with three extra fields retrieved while reading and writing the formalization. Making choices happens quite often as the internal data structures are broadly defined, being “implementation dependant”. We reference the execution contexts in the Skel implementation. This structure is stored in a mapping in the state itself, relating locations to execution contexts.

Execution Context Stack In the specification, the execution context stack is defined textually as:

ECMAScript

> The execution context stack is used to track execution contexts. The running execution context is always the top element of this stack. A new execution context is created whenever control is transferred from the executable code associated with the currently running execution context to executable code that is not associated with that execution context. The newly created execution context is pushed onto the stack and becomes the running execution context.

In Skel, we do not have any implementation of this data structure, so we chose to represent the type stack as an unspecified type. We also define the type helper functions to manipulate the stack. The execution contexts held in the stack have a status field. In case of an execution context suspension, some algorithmic steps of the specification may be suspended too. In this case, before resuming, the latter suspended execution context resumes the suspended code, possibly altering the state, and then evaluates the algorithmic steps following its resumption. This mechanism is quite complex and, in the context of the thesis, causes problems in handling the execution context stack, as continuations manipulate the normal LIFO behavior of the stack data structure. Indeed, in the following text, we present sentences of the specification which hint at the more complex control flow of the algorithmic steps.

ECMAScript

> Evaluation of code by the running execution context may be suspended at various points defined within this specification. Once the running execution context has been suspended a different execution context may become the running execution context and commence evaluating its code. At some later time a suspended execution context may again become the running execution context and continue evaluating its code at the point where it had previously been suspended. Transition of the running execution context status among execution contexts usually occurs in stack-like last-in/first-out manner. However, some ECMAScript features require non-LIFO transitions of the running execution context.

We plan to change the actual formalization, including the work shown in Section 3.5.

An execution context can be seen as a delimited code section, which can yield, producing a non-linear behavior of the executable specification code. In Section 4.1.6, we will describe how these non-LIFO manipulations of the execution context stack can invalidate some of our ES formalization features, such as a simple function call with at least one argument.

Environment Records The Environment Record is a specification type used to define the association of Identifiers to specific variables and functions based upon the lexical nesting structure of ECMAScript code. Usually, an Environment Record is associated with some specific syntactic structure of ECMAScript code, such as a *FunctionDeclaration*, a *BlockStatement*, or a *Catch* clause of a *TryStatement*. Each time such code is evaluated, a new Environment Record is created to record the identifier bindings created by that code.

This record, merely a specification mechanism, cannot be accessed and modified by programs, as it is considered internal to an interpreter. The specification does not provide directions on how to implement this data structure as it is unnecessary, giving free will in describing it. As the project's goal is a loyal formalization of the specification, we tried to collect every possible information about the execution contexts in the document to provide a reference implementation. We must say that the specification provides a classification of this record.

This record can be a *Declarative Record*, used to define the effect of ECMAScript language syntactic elements such as FunctionDeclarations, VariableDeclarations, and Catch clauses that directly associate identifier bindings with ECMAScript language values (Null, Undefined, Boolean, Number, BigInt, String, Symbol, and Object). A Declarative Record can be a *Function Environment Record*, storing information about bindings for the top-level declarations within that function and establishing a new `this` binding. Also, it captures the state necessary to support `super` method invocations, or a *Module Environment Record*, containing the bindings for the top-level declarations of a *module*. This work focused more on the semantics of evaluating the scripts rather than JS modular code and modules. Hence, we did not represent the latter category. Then, the record can be an *Object Environment Record* or a *Global Environment Record*. For the first category, the record defines the effects of some ECMAScript elements that associate identifier bindings with the properties of some object.
An example can be the relation between Object Environment Records and the *With-*

Statement. This Environment Record type associated with a `with` statement can provide its bound object as an implicit argument of a function call. This argument is the `this` value of the called function.
The second record type is used for the script's global declarations. It may be prepopulated with identifier bindings and includes an associated global object, the standard library of JavaScript.

The Environment Records have a field called [[OuterEnv]], a mechanism that enables the hoisting feature—default behavior of moving all declarations to the top of the current scope. This mechanism is similar to accessing properties via the prototype chain, a link between objects that establishes a hierarchy between objects and prototypes that define them.

The record we design is a record containing all the fields of all the possible categories of environment records. Then, we can identify the type of a given environment record by defining an extra field providing the type. We decided not to rely on the typical duck-typing[3] of languages such as JavaScript, which is not convenient because of Skel being a strongly typed language. Non-mandatory fields are defined as option-type fields, according to our understanding of the specification. The record is too big to be reported in this thesis. It can be inspected at the following URL: `https://gitlab.inria.fr/skeletons/jskel/-/blob/matches/semantics/js.sk#L4479`.

We also provide getter and setter functions that manipulate the mapping between locations (numbers) and environment records in the state. In the Skel implementation, we always reference the environment records, resolving the mapping only if we need to manipulate the referenced record's contents.

In the record, we define a field for each one of the internal methods. These are higher-order fields. An instance of the Environment Record has each function instantiated with its location, allowing one to apply the method by only passing the arguments defined in the specification. For example, given an environment record `er`, if we want to call its internal method `_Method_` with parameter `par`, we do `er._Method_ par`. Implicit in the call is the location of the record `er`.

Realm Records This record consists of a set of intrinsic objects, an ECMAScript global environment, all of the ECMAScript code executing within the scope of that global environment, and some other associated resources.

3. "If it walks like a duck, and it quacks like a duck, then it must be a duck."

The intrinsic objects are the standard library of the language. They are the prototype of all the possible features that a custom-defined object might have. We design the intrinsic object field as a record in which each field is labeled after the specification name of each intrinsic object (`%ObjectName%`) and has the object's location as its value. Then, the record contains a reference to the environment of the global object, the first environment created while instantiating program interpretation. The ECMAScript code field has the AST of the program currently being evaluated. Then there are additional pieces of information, such as the [[TemplateMap]], which we do not currently consider, and the [[HostDefined]] field, providing some additional features to the language, extending what the standard defined. For example, NodeJS implements the server language extension as host-defined features. These custom features are more related to implementations of the language rather than the standard itself. We do not consider this field in our formalization.

In the implementation, we map locations to Realm Records, as with the other records stored in the state. Then the state field is a map. Realm Record does not have internal methods, but it is often manipulated. For that, we design a set of accessors and modifiers to manipulate the actual Realm Record stored in the state.

Script Records The Script Record is a specification mechanism containing information about the evaluation of a script. It is implemented similarly to the other records. It references the Realm Record in which the script is evaluated, the environment in which the top-level bindings of the script are stored, the ES AST representing the program, and some host-defined additional information related to the script evaluation. Its implementation goes straightforwardly with the one of the Environment Record.

Objects ECMAScript objects are not fundamentally class-based such as those in C++, Smalltalk, or Java. Instead, objects may be created in various ways, including via a literal notation or via constructors. These create objects and execute code that initializes all or part of them by assigning initial values to their properties. Each constructor is a function that has a property named "prototype", which is used to implement prototype-based inheritance and shared properties.

Like the Environment Records, also the objects are categorized. We implement the object type as a record containing all the object types' fields and properties, turning on and off an object field via option-like type values. All the internal methods store the

object's location, providing uniformity to the overall formalization approach. We again do not rely on the duck-typing by adding an extra field describing the type of the instantiated object. In Figure 4.2, we present the object's internal representation `{n : 42}`. This output is produced after interpreting the program via the OCaml interpreter generated from Skel. The field `_ObjectType_` shows object type, an `OrdinaryObject`. The binding of the property `n` to `42` is the first element of the property list—field `_Properties_`. Each internal method of the object is instantiated with a `this` value, a reference to the object itself.

We handle the objects stored in the state as we did with the previous records, making the state field _Objects_ a mapping between locations and the actual type encoding objects.

In the artefact, we provide a complete description of all the categories of objects. We do not list them here, as presenting them does not provide additional information on the approach to formalizing the ES document. Nevertheless, the type definition is available at `https://gitlab.inria.fr/skeletons/jskel/-/blob/matches/semantics/js.sk#L1714`.

A Local Environment The local environment is not defined in the document but is necessary for our approach. The specification style is pseudo-algorithmic. It often happens that a new name not existing in the upper scope of an algorithmic step is created inside an inner scope and then propagated to the top-level contexts. The specification implicitly defines a hoisting method in the context of single ES steps. We add to the state a local environment to hoist these bindings outside the scope of an algorithmic step. In Figure 4.3, we provide an example of these algorithmic steps. In the first step, a new name bar is set to false if foo is true. In the second step, the bar is inspected, but in the case of foo being `false`, generally, the specification considers the bar value `undefined`. We use the local environment first to set all the inner-scope instantiated names as undefined. Then we manipulate these names in this special context, outside the skeletal semantics' one.

```
{__ThisMode__ = VNone; __StringData__ = VNone; __Strict__ = VNone;
 __SourceText__ = VNone; __SetPrototypeOf__ = <fun>;
 __ScriptOrModule__ = MN_Null; __Realm__ = VNone;
 __ProxyTarget__ = VNone; __ProxyHandler__ = VNone;
  __InitialName__ = VNone; __GetPrototypeOf__ = <fun>;__Prototype__ =
  MN_NotNull (ObjectLocation_Intrinsic IntrinsicObjectPrototype);
 __PreventExtensions__ = <fun>; __ParameterMap__ = VNone;
 __OwnPropertyKeys__ = <fun>; __Name__ = VNone;
 __IsExtensible__ = <fun>;__IsClassConstructor__ = VNone;
 __HomeObject__ = VNone; __HasProperty__ = <fun>;
 __GetOwnProperty__ = <fun>; __GeneratorState__ = VNone;
 __GeneratorContext__ = VNone; __GeneratorBrand__ = VNone;
 __FormalParameters__ = VNone; __Extensible__ = VSome T;
 __Environment__ = VNone; __Env__ = VNone; __ECMAScriptCode__ = VNone;
 __Delete__ = <fun>; __DefineOwnProperty__ = <fun>;
 __ConstructorKind__ = VNone; __Construct__ = VNone; __Call__ = VNone;
 __BoundThis__ = VNone; __BoundTargetFunction__ = VNone;
 __BoundArguments__ = VNone; __AsyncGeneratorState__ = VNone;
 _Properties_ =
  List.(::)
   ({_PropertyKey_ = Str "n";
     _Descriptor_ =
      {__Writable__ = VSome T;
       __Value__ = VSome (Numeric (Number (Float 42.)));
       __Set__ = VNone;__Get__ = VNone; __Enumerable__ = VSome T;
       __Configurable__ = VSome T}},
   []);
 _ObjectType_ = OrdinaryObject; _O_Set__ = <fun>; _O_Get__ = <fun>;
 _ConstructorSteps_ = VNone; _AdditionalSlots_ = List.[]}
```

Figure 4.2: Internal representation of the object `{n : 42}`.

1. If foo is `true`
   - Let bar be `false`
2. If bar is ...

Figure 4.3: Example of algorithmic steps in which a local environment is necessary.

```
type completionType=
| Normal
| Break
| Continue
| Return
| Throw
```

```
type completionValue<a>=
| Ok a
| Abruption maybeEmpty<value>

type completionTarget:=
maybeEmpty<string>
```

```
type completionRecord<a> =(_Type_: completionType,
                           _Value_: completionValue<a>,
                           _Target_: maybeEmpty<string>)
```

Figure 4.4: Skel Formalization of ES Completion Records

4.1.3 Completion Record and the ECMAScript Error Handling (?!) monad

Most ES operations do not directly return values, they return *completion records*[4] instead. A Completion Record describes the runtime propagation of values and control flow. This record is composed of three fields: a kind (either a result, a break out of a loop, a return of a function, or an exception being thrown), an optional value, and in the case of a break or continue, a target.

In theory, a completion record should only hold ES language values (Null, Undefined, Boolean, Number, BigInt, String, Symbol, and Object) or be empty. In practice, it is used to return many other constructions.[5] If we consider the `getValue(V)` abstract operation, the completion record given as input can hold either a Value or a Reference in its `_Value_` field. Many such examples litter the specification. Hence, we consider completion records to be polymorphic in what their "value" field holds. We declare such completion records as `type completionRecord<a>`. Figure 4.4 defines the types corresponding to the contents of this record: the `completionType` holds the kind of the record, and the `completionValue` is either an `Ok` polymorphic constructor that contains the value of a non-abrupt computation or an abruption. An ES abrupt Completion Record is one whose type is not **normal**. For instance, a **throw** record has an exception

4. https://tc39.es/ecma262/2021/#sec-completion-record-specification-type
5. In early 2022, the specification changed to accept any value and not just language values, see https://github.com/tc39/ecma262/pull/2547

```
type out<a> =
| Success completionRecord<a>
| Anomaly anomaly
```

```
type anomaly =
| AbruptAnomaly completionRecord<()>
| StringAnomaly string
| NotImplemented
```

Figure 4.5: Out and Anomaly Declarations

object as completion value, a **return** record has an optional value, and a **break** record has `empty`. Due to the aforementioned specification issues where non-values may be returned, we store the optional value in a separate constructor to be able to return it independently of the completion type. Finally, the `completionTarget` holds the optional string representing the target, typically used for a `break` to a label. An ES completion record for values has type `completionRecord<maybeEmpty<value>>`.

We next define a type `out` (Figure 4.5) composed of two constructors, `Success` for *successful computation*s and `Anomaly` for anomalies, which intuitively corresponds to a failure of the specification, e.g., when an assertion does not hold. A successful computation is one that returns an ES completion record, either `Normal` or `Abrupt`. The `Anomaly` constructor captures incorrect computations. In the specification, anomalies can be raised by assertion failures or when it is explicitly written that an abstract operation call must not return an `Abruption`. The specification authors informally guarantee that these failures never occur.

The `Anomaly` constructor is of type `anomaly`. It has three type constructors: `AbruptAnomaly` holding a completion record in case an evaluation returns an unexpected abruption, `StringAnomaly` that contains textual information about an anomaly—when an assertion of the specification is broken or when an implementation-dependent operation fails, and the `NotImplemented` signaling that we have not yet implemented some feature.

ES abruptions are propagated through an abstract method called `ReturnIfAbrupt`. Basically, this method gets a completion record and either returns the value of the `_Value_` field in case of a normal completion, or it propagates the abruption.

In cases of an abstract operation or a recursive evaluation, the specification uses the prefix `?` to indicate that `ReturnIfAbrupt` has to be applied to the resulting completion. This operator is a monadic bind for a variant of the classic exception monad. The other operator used in the specification is `!`: it behaves similarly to `?` on a normal result, but it asserts an abruption cannot occur. We thus model it as transforming an abrup-

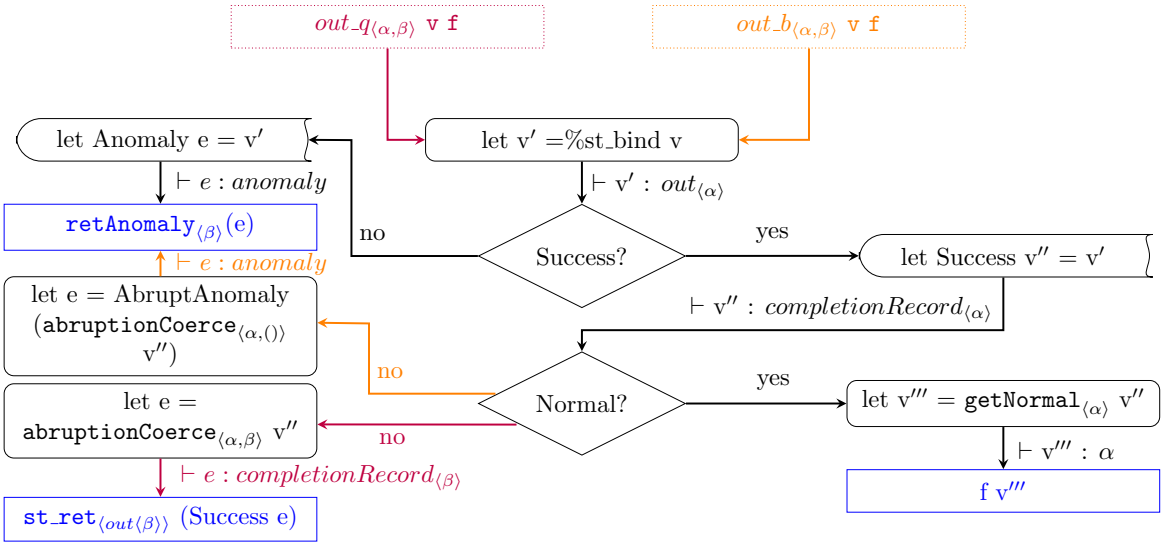

Figure 4.6: The Model of `?` and `!`.

tion into an anomaly. These behaviors are reflected in the flow-chart presented in Figure 4.6, presenting respectively `out_q`$_{\langle\alpha,\beta\rangle}$(red arrows) for the `?`, and `out_b`$_{\langle\alpha,\beta\rangle}$(orange arrows) for the `!`.

We define them as the monadic binders of the combination of `st` and `out`. We put the state monad outside as we want to return a state in the case of an exception.

```
val out_q<a,b>: st<out<a>> → (a→st<out<b>>) → st<out<b>>
val out_b<a,b>: st<out<a>> → (a→st<out<b>>) → st<out<b>>
val out_ReturnIfAbrupt<a,b> : out<a> → (a → st<out<b>>) → st<out<b>>
binder ?out = out_q
binder !out = out_b
```

We also define getters (`getNormal`, `getAbrupt`, ...) to extract values from a `completionRecord`, and return functions (`retAnomaly`, `retAbrupt`, `retNormal`, ...) for successful and anomaly computations. Note the use of the `abruptionCoerce` operation that is only defined for completion records that are abruptions. It is the identity, but it changes the type parameter of the completion record. What follows shows how faithful is the translation of these types of algorithmic steps in Skel. The annotation τ_{bar} is the type of `bar`.

```
type cf<a> =
| ReturnControl a
| ContinueControl

val cf_ret<a> (v:a) → cf<a> =
 ReturnControl<a> v

val cf_cont<a> : cf<a> =
 ContinueControl<a>
```

```
val cf_bind<a> : cf<a> →
                 (() → cf<a>) →
                 cf<a>
binder @cf := cf_bind

val cf_res<a> : cf<a> → (a → a) →
                a
binder <cf := cf_res
```

Figure 4.7: The controlFlow type with binders and setters

1. let bar be ? AbstractOperation(foo)

2. Return bar

```
let bar =?out abstractOperation(foo) in
retNormal<τbar> bar
```

4.1.4 A Control-Flow monad

As said earlier, the ES specification uses imperative control flow, such as returning in the middle of an algorithm. In Figure 4.7, we introduce the `type cf<a>`. It represents computations that can either continue or that terminate with a result of type `a`. It comprises two constructors: `ReturnControl`, to return a result of type `a`, and `ContinueControl` to continue the execution.

We define a monadic binder `cf_bind`, a function `cf_res` to extract the value from a `cf<a>` term, a function `cf_ret` to break the control-flow and return a value, and a function `cf_cont` to signal the execution should continue.

We illustrate in Figure 4.8 the translation of algorithmic steps in Skel, with and without the control-flow monad. We recall that we use the forms `let True = ... in ...` to test whether a value is `true`, and `let False = ... in ...` to test whether it is `false`. We also write `foo;@ bar` for `let _ =@ foo in bar`. Despite the slight overhead in notation, the control-flow approach is closer to the algorithmic steps and can be consistently applied to deal with the typical case of a conditional that returns without an else branch. One can achieve a similar behavior without the control flow monad at the

1. If foo is `true` Return `1`
2. If bar is `true` Return `2`
3. If baz is `true` Return `3`
4. Return `4`

```
  (*no control-flow monad*)
branch let True = foo in 1
or     let False = foo in
       branch let True = bar in 2
       or     let False = bar in
              branch let True = baz in 3
              or     let False = baz in 4
              end
       end
end
```

```
  (*control-flow monad*)
let result =<cf
   branch let True = foo in cf_ret<int> 1
   or     let False = foo in cf_cont<int>
   end;@cf
   branch let True = bar in cf_ret<int> 2
   or     let False = bar in cf_cont<int>
   end;@cf
   branch let True = baz in cf_ret<int> 3
   or     let False = baz in cf_cont<int>
   end;@cf
   cf_ret<int> 4
in result
```

Figure 4.8: Code with and without Control Flow Monad.

cost of nested branching. It is possible to avoid the nesting of branching by hoisting all the branches at top-level. This results in the following code.

```
branch let True = foo in 1
or     let False = foo in let True = bar in 2
or     let False = foo in let False = bar in let True = baz in 3
or     let False = foo in let False = bar in let False = baz in 4
end
```

The reason previous conditions are repeated is because there is no guarantee that the evaluation of a branching will consider branches in declaration order. In addition, using a collecting semantics (where all branches are considered) would give the wrong result if we did not restate all conditions.

We define `st<cf<out<a>>>` type as the combination of the three monadic types, and accordingly, the definitions of the binders and return functions as `bind` (@), `cf_out` (<), `cont`, and `ret`. Our general approach is to encapsulate all the algorithmic steps in this type, returning an `st<out<a>>` at the end of every function that may raise an exception. To this end, we start each algorithm with `let result =<` to enter the full monad with control, and we exit the control monad at the end of the algorithm with `in result`. Figure 4.9 gives a variant of Figure 4.8 with exceptions. Notice that the only thing that changes is the first step and the use of the appropriate monadic binders for the type `st<cf<out<a>>>`.

Despite the closeness to the algorithmic steps, the latter figure, in lines 3, 5, and 7, shows redundancy of code in the `or` branches. Indeed, each time the condition is not satisfied, we have to explicitly state that the evaluation continues. To avoid doing so, we define two boolean binder-like functions, `ifTrue` and `ifFalse`, for introducing partiality in branchings. The `ifTrue` binder, denoted as `@t`, evaluates the rest of the branch when given the `True` value and directly returns a `ContinueControl` when given the `False` value. The following code applies the `ifTrue` binder to the example in Figure 4.9.

```
let result =<
   branch foo;@t throw<int> fooError end;@
   branch bar;@t ret<int> 2 end;@
   branch baz;@t ret<int> 3 end;@
   ret<int> 4
in result
```

1. If foo is `true` throw `FooError`
2. If bar is `true` Return 2
3. If baz is `true` Return 3
4. Return 4

```
let result =<
    branch let True = foo in throw<int> fooError
    or     let False = foo in cont<int> () end;@
    branch let True = bar in ret<int> 2
    or     let False = bar in cont<int> () end;@
    branch let True = baz in ret<int> 3
    or     let False = baz in cont<int> () end;@
    ret<int> 4
in result
```

Figure 4.9: Variant of the Figure 4.8 with Exceptions.

4.1.5 A Real Example in Skel

In this section, we show the effectiveness of the design choices introduced in the previous sections. We proceed by presenting the formalization of the `GetValue` abstract method as an example. This method, presented at the top of Figure 4.10, is one of the most used methods in the specification, as it is often called after the evaluation of syntactic constructors of the language. It takes V as input, a completion record containing either a value or a reference, and returns a value as output. In a nutshell, if V is a value, `GetValue` returns it, and if it is a reference, `GetValue` acts like a binding resolver to obtain a primitive value from an Object or an Environment Record.

The reference[6] type is a record that contains four fields, including the `[[Base]]` field that holds an environment record or an ES value and three other fields not relevant here. Our mechanization in Skel specifies references as records too.

We now describe step-by-step how we formalize this method (the code is collected as a single function at the bottom of Figure 4.10). The type mixing values and references is the specified type `valref` defined as `| Val value | Ref reference`. The argument of `GetValue` thus has type `out<valref>`. We highlight the code that does not

6. `https://tc39.es/ecma262/2021/#sec-reference-record-specification-type`

1. ReturnIfAbrupt(V).
2. If Type(V) is not Reference, return V.
3. If IsUnresolvableReference(V) is **true**, throw a **ReferenceError** exception.
4. If IsPropertyReference(V) is **true**, then
   a Let base be V.[[Base]].
   b Let baseObj be ! ToObject(base).
   c Return ? baseObj.[[Get]](V.[[ReferencedName]], GetThisValue(V)).
5. Else,
   a Let base be V.[[Base]].
   b Assert: base is an Environment Record.
   c Return ? base.GetBindingValue(V.[[ReferencedName]], V.[[Strict]]).

```
1  (*1*) let v =%returnIfAbrupt v in
2  (*2*) branch valref_Type(v,T_Ref);@f let Val v = v in ret<value> v end;@
3  (*N*) let Ref v = v in
4  (*3*) branch isUnresolvableReference(v);@t
5              throw<value> referenceError end;@
6  (*4*) branch let True = isPropertyReference(v) in
7        (*a*) let R_Val base = v.base in
8        (*b*) let baseObj =! toObject(base) in
9        (*N*) let baseObj =/o baseObj in
10       (*c*) let thisVal =? getThisValue(v) in
11       (*c*) let r =? baseObj._Get_(v.__ReferencedName__, thisVal) in
12       (*c*) ret<value> r
13 (*5*) or     let False = isPropertyReference(v) in
14       (*a*) let base = v.base in
15       (*b*) assert_true<(reference,type_ref)> ref_Type (base, T_R_ER);@
16       (*N*) let R_ER base = base in let base =/er base in
17       (*c*) let r =? base._GetBindingValue_(v.__ReferencedName__,
18             v.__Strict__) in
19       (*c*) ret<value> r
20 end
```

Figure 4.10: The ECMAScript's `GetValue(V)` and its Skel formalization

correspond to the ES algorithmic steps, i.e., code that is an overhead of the Skel formalization. It is necessary for typing reasons (extracting a value from a variant type) or

transforming a heap location into its contents.

The first step of the abstract method applies `ReturnIfAbrupt` on V. In case it is an abruption, the method propagates the completion record to the caller. Otherwise, it extracts the `completionValue`. To model this, we use the `/\%/returnIfAbrupt` monadic binder. We could directly write `let v =@ returnIfAbrupt<(valref,value)>(v)`, but this requires specifying the polymorphic type components, which is not necessary for binders as they are inferred. We are working on extending the type inference to make polymorphic annotations unnecessary and thus be closer to the specification. If V is a normal completion, the newly bound V will have type `valref`.

The next ES step inspects the type of V. If it is not a reference, hence a value, we return it. Otherwise, the `@f` implicitly continues the execution. Then, in Skel 's line 3, we can safely extract v from the `Ref` type constructor. Now `v` has type `reference`.

Step 3 follows the same pattern as step 2: An *unresolvable* reference is one that has `Undefined` or `Null` as `[[Base]]`. In this case, a `ReferenceError` is thrown. In Skel, we proceed straightforwardly. If the reference is unresolvable, we throw a `referenceError` of type value. The `throw<a>` function takes an error object constructor as an argument. Then, it returns an abrupt `completionRecord` that contains as a `completionValue` a reference error object.

In case the reference is resolvable, step 4 inspects whether V is a *property reference*. A property reference is a reference that has a non-null, defined base value of type `value`. Step 4.a.assigns the base value stored into the reference to the variable V. In line 6 of the Skel formalization, we access the record field. In the same line, we make an inline pattern matching, expecting the base value to be of type `value` (`R_Val`). Now base is a primitive value, meaning one of type `Boolean`, `String`, `Symbol`, `BigInt`, or `Number`. Then, line 7 describes Step 4.b. The primitive value in base is cast to an object. The method `ToObject`, transforms a primitive value into an object, raising an exception when the value is `Undefined` or `Null`. This operation call is prefixed by `!`, meaning that this method should never raise an exception. Indeed, dealing with primitive objects prevents from getting an exception[7] as a result of the cast operation. In Skel, the result of `toObject` is an object location. In line 8, we take the object value from the state with the monadic binder `/o`. Finally, Step 4.c. returns the result of the baseObj's `[[Get]]` object internal method applied to a name representing the referenced value's name and to a value resulting from the call to the `GetThisValue` method.

7. In case of an exception, a specification anomaly is then raised.

In Skel, we split this step into three. First, the method `getThisValue` is called on the reference `v`, assigning its result to the variable `thisVal`. Note that we make explicit the call to `getThisValue`, as Skel requires function application to consider fully computed terms as arguments. The call to `getThisValue` is not pure as the assertion there[8] is not captured by typing (although we have checked that the reference is indeed a property reference at step 4). We thus use the `?` binder, in that case, to extract the pure value or propagate the abruption, if any. Second, once `baseObj` is set, we call its `_Get_` internal method. This method takes `v.__ReferencedName__` and `thisVal` as arguments. The call is prefixed by `?`, meaning, again, that if an abrupt result happens, it is propagated to the caller. Finally, we return the result of the call.

If V is not a property reference, then base must be an Environment Record. The Skel else branch, namely Step 5., follows the structure of Step 4. Notice that in 5.b, there is an assertion that we capture in line 14. It can be read as *"assert that is true that the reference base-value `base` is an Environment Record"*. The `ref_Type` is a function that takes a tuple of type `(reference, type_ref)`, returning a boolean. The `type_ref type` constructor `T_R_ER` represents references with Environment Records as base values. The `assert_true` is a function that takes as arguments a function `f` and its arguments `a`. It propagates an anomaly when the application `f a` is `False` or continues to compute in case of a `True` judgment.

As with most of the algorithms in the Skel mechanization, we need to return a result that is out of the control-flow monad type. The following piece of code thus surrounds the whole algorithm.

```
let result =< (* GetValue's algorithmic steps *) in
result
```

4.1.6 Current Status

We defined a three-steps entry-point to the mechanization. First, we create the environment in which the code will be evaluated by initializing the interpretation environment[9]. This operation creates the *Realm*, a record to which all the evaluated ECMAScript code is associated, the *Intrinsic Objects*, the main *Execution Context*, the *Global Envi-*

8. https://tc39.es/ecma262/2021/#sec-getthisvalue
9. https://tc39.es/ecma262/2021/#sec-initializehostdefinedrealm

ronment, and the *Global Object*, which can be considered as the standard library of JS. Second, we parse the script[10], and finally, we evaluate the script[11]. To execute these three steps, it is required a significant implementation of the specification, mostly from [20, Chapters 6 to 10, 18, 19, 20]. Once the runtime environment of the specification is defined, extending the Skel mechanization comes strightforward.

Concerning the language itself, we defined in Skel a significant subset of ES Syntax and its evaluation. Unfortunately, no existing parser of JS provides a faithful representation to ES Syntax. We thus had to choose between implementing our own parser or translating the AST provided by on-the-shelf parsers. We chose the latter. More precisely, we use a parser that conforms to the *SpiderMonkey*'s Parser API[12], which is followed by all parsers we have found. We chose the Flow Parser[13] library because it is written in OCaml, which is the language we can instantiate our interpreter into, and because it is used to manipulate JS in industrial setting. When instantiating the interpreter, we define transformations that, given a Flow AST, produces a well-typed ES AST.

We implemented most of the *Statement*s, and *LexicalDeclaration*s([20, Chapter 14]). Indeed, these two syntactic production define what a script is([20, Section 16.1]).

For what concerns the *Expression*s([20, Chapter 13]) of the language, we formalized most of the chapter. We are still working on the semantics of some *CallExpression*s and of the function definitions([20, Sections 13.3, 15.5, 15.6, 15.8, 15.9]), as their evaluation explicitly manipulates continuations of the interpreter code. This prevents us from start testing interpreter via the ES262 test suite, as function calls are required for testing. Nevertheless, understanding how the function calls are specified led to the discovery of some places where the manipulation of execution contexts were not properly described or handled. [14]

Intuitively, an ECMAscript function call first evaluates the actual arguments, binding them to local variables, before executing the core of the function in this new environment. The algorithmic steps for processing all the inputs of a function and then binding them in the function environments are defined in the internal method

10. https://tc39.es/ecma262/2021/#sec-parse-script
11. https://tc39.es/ecma262/2021/#sec-runtime-semantics-scriptevaluation
12. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API
13. https://github.com/facebook/flow
14. https://github.com/tc39/ecma262/issues/2400, https://github.com/tc39/ecma262/issues/2409

(a) For each element `E` in `list`, do

  (i) Perform ? `Yield`(`E`)

(b) Return **undefined**

Figure 4.11: yieldable abstract closure

`FunctionDeclarationInstantiation`[15]. In step 24 of this method, there is a call to `CreateListIteratorRecord`[16] with `argumentsList` as a parameter, a list of the arguments of the function. An `abstract closure` **closure**, shown in Figure 4.11, is then defined. It corresponds to a delimited computation that yields the next argument each time it is resumed (the *?* in the code is a monadic binders that intuitively propagates an exception if one happens). The specification then creates an `iterator` through the call of the method `CreateIteratorFromClosure`[17]. Parameters of the call are **closure**, that is a piece of code that will be associated to the iterator **iterator**, and `%IteratorPrototype%`, prototype of the object that the method creates. The object is extended with some extra *internal slots* (step 3-`OrdinaryObjectCreate`[18]) `[[GeneratorState]]`, `[[GeneratorContext]]`, and `[[GeneratorBrand]]`. A new execution context is then created, similar to using `Run` in Section 3.5, and `GeneratorStart`[19] is called in this context. In the particular case of function calls, this method sets a call to **closure** as a resumption handler. The iterator is then suspended, and `CreateIteratorFromClosure` returns it. Then, `CreateListIteratorRecord` binds it to **iterator** and it returns an iterator record with **iterator** as the iterator object and `%GeneratorFunction.prototype.prototype.next%` as next method of the (suspended) iterator. Back to `FunctionDeclarationInstantiation`, the next step is the call to *IteratorBindingInitialization*[20]. This uses the iterator to repeatedly call its *next* method, which itself calls `GeneratorResume`[21]. This makes a context switch that puts the generator context as the running execution context, and it resumes it by calling the closure stored in the context code evaluation state. This is similar to

15. `https://tc39.es/ecma262/#sec-functiondeclarationinstantiation`
16. `https://tc39.es/ecma262/#sec-createlistiteratorRecord`
17. `https://tc39.es/ecma262/#sec-createiteratorfromclosure`
18. `https://tc39.es/ecma262/#sec-ordinaryobjectcreate`
19. `https://tc39.es/ecma262/#sec-generatorstart`
20. `https://tc39.es/ecma262/#sec-runtime-semantics-iteratorbindinginitialization`
21. `https://tc39.es/ecma262/#sec-generatorresume`

```
module IntMap = Map.Make (Int)

module rec T : sig
  type bigInt = Big_int.t
  and float = Float.t
  and int = Int.t
  and 'a intMap = 'a IntMap.t
  and 'a list = 'a List.t
  and string = String.t
end = T
```

Figure 4.12: TYPES module instantiation

calling `resume` in Section 3.5. Running the closure results in the call to the `Yield` in it, providing the next argument.

Understanding how the function calls are specified led to the discovery of some places where the manipulation of execution contexts were not properly described or handled[22,23]. This resulted into some clarifications and bug fixes, such as the creation of a new execution context for the generator.[24] Links to the specification above are to the current version where the clarifications are included. To read the 2021 version, replace `https://tc39.es/ecma262/` with `https://tc39.es/ecma262/2021/`.

4.2 Interpreter Evaluation

4.2.1 Interpreter Instantiation

To instantiate an interpreter generated by `necroml`, we need to provide an implementation to the unspecified types and terms.

Regarding the unspecified types, although the ES specification in Skel consists of 15k LOC, only 6 types need a definition. We show the instantiation of the `TYPES` module in Figure 4.12. Notice that we could have left unspecified four data types out of six, by writing a library defining the `list` and `intMap` in Skel.

22. `https://es.discourse.group/t/execution-context-suspension-and-resumption/756`
23. `https://github.com/tc39/ecma262/issues/2400`
24. `https://github.com/tc39/ecma262/pull/2398`

Concerning terms, we have 450 of these unspecified elements in our skeletal semantics. We instantiate some of the unspecified functions—the ones that the interpreter uses-, leaving the others returning `NotImplemented`. Finally, we define a parsing method to transform a JavaScript program into an ECMAScript AST. To this end, we rely on the Flow Parser AST and convert the result to the (much more complex) AST used by ECMAScript. We are then ready to run examples.

As we said in Section 4.1.6, we have gradually implemented core functionalities of the ES to crash test our specification language Skel and see if it is expressive enough to represent such complex semantics faithfully.

4.2.2 Evaluation

We propose two ways of evaluating this work. The first one is to compare our work with previous ones (Framework Comparison), and the second is to describe what JS programs we can execute (Program Execution).

4.2.2.1 Framework Comparison

One of the goals of this work is to show visual closeness and maintainability of the proposed formalization. We compare our work with $\mathbb{K}_{\text{JS}}$ [56] and JSCert [6] by considering the formalization of of the `GetValue` internal algorithm. Note that both these formalizations are based on ECMAScript 5.1, an older version of the standard. The ECMAScript standard tripled in size since then and its algorithms are much more complex, including explicit continuation manipulation.

$\mathbb{K}_{\text{JS}}$ is a set of rewriting rules representing the ES semantics. This means that given an internal method, there are a subset of whole rewriting rules set, matching the method's behaviour. To simplify, we can say that the left hand side component of the rule is rewritten into the right hand side when the conditions allow it. The correctness of $\mathbb{K}_{\text{JS}}$ is attested by a great coverage of the ES262 test suite. Consider the `GetValue` algorithmic steps in ECMAScript 5.1[25]. One can notice the differences between the method of the 2021 standard and the one of version 5.1. The old rule can be split in four cases defined by the types that the `GetValue`'s parameter `V` can take. Indeed, `V` can be either a *reference*, an *object*, the *undefined* value, or a *primitive* value. In Figure 4.13, we present the $\mathbb{K}_{\text{JS}}$ implementation. The 4 rules concisely cover all the cases.

25. `https://262.ecma-international.org/5.1/#sec-8.7.1`

```
rule GetValue(@Ref(E:Eid,N:Var, Strict:Bool)) =>
     GetBindingValue(E,N,Strict)
rule GetValue(@Ref(O:Oid,P:Var, _)) =>  Get(O,P)
rule GetValue(@Ref(Undefined, P:Var, _ )) =>
     @Throw(@ReferenceError("GetValue",P))
rule GetValue(@Ref(B:Primitive,P:Var, _ )) =>
     GetPrimitive(B,P)
```

Figure 4.13: GetValue 5.1 written in the $\mathbb{K}$ framework.

In case of `V` being a reference, the GetValue is rewritten in a GetBindingValue. If it is an object, the definition is rewritten to an object `Get`. In the latter two cases, instead of using the method `IsUnreasolvableReference` and `HasPrimitiveBase`, the authors define two rules, one in which `V` is Undefined, which corresponds to an unresolvable reference, and the other by matching the input parameter with a Primitive label, representing a subtype of the ES value type.

We argue our Skel formalization is better for the following reasons. First, we are very close to the textual description of the standard. In fact, when translating it to Skel, we do not need to fully understand its behavior. This suggests that an automated translation, as done in [59], could be achieved. Second, the formalization has not been updated since 2015. Third, the formalization in itself is not easily manipulated by external tools. In contrast, our Skel definition can be used to generate a Coq description, which is not possible with $\mathbb{K}$.

We show a part of the formalization of `GetValue` in JSCert in Figure 4.14.[26] The formalization is a Coq shallow embedding of ES, in pretty-big-step semantics style [15]. As said before, the two main issues with JSCert are maintainability and usability. Regarding the latter, the semantics is too big to be used in the context of Coq induction and inversion (Coq runs out of memory). Using a deep embedding (where the recursive evaluation of the language semantics is not captured using Coq's induction) would help avoid the issue, but doing so requires rewriting the semantics. As Coq descriptions are not easily manipulated by outside tools, this motivated us to redo the semantics in a framework where this issue could be circumvented, hence the current work.

To conclude, neither $\mathbb{K}_{\text{JS}}$ nor JSCert are easily maintainable, as their approaches are too much embedded in the tools they are using. By having a simple but powerful

26. Full code at `https://github.com/jscert/jscert/blob/master/coq/JsPrettyRules.v#L5112`

```
(** Get value on a reference (returns value) (8.7.1) *)

| red_spec_ref_get_value_value: forall S C v, (* Step 1 *)
    red_spec S C (spec_get_value v) (ret S v)

| red_spec_ref_get_value_ref_a: (* Steps 2 and 3 *)
    forall S C r (y:specret value),
      ref_is_unresolvable r ->
      red_spec S C (spec_error_spec native_error_ref) y ->
      red_spec S C (spec_get_value r) y

| red_spec_ref_get_value_ref_b_has_primitive_base:
    (* Steps 2 and 4 *)
    forall v S C r o1 (y:specret value),
      ref_is_property r ->
      ref_base r = ref_base_type_value v ->
      ref_has_primitive_base r ->
      red_expr S C (spec_prim_value_get v (ref_name r)) o1 ->
      red_spec S C (spec_get_value_ref_b_1 o1) y ->
      red_spec S C (spec_get_value r) y
```

Figure 4.14: GetValue 5.1 in JSCert

specification language, we are able to systematically write code that is very close to the standard and easy to maintain. We migrated from ECMAScript 2020 to ECMAScript 2021 in less that a week of work, by only changing the algorithmic steps that have been modified.

4.2.2.2 Program Execution

After instantiating the interpreter, we started to test it gradually. We show, in Figure 4.15, an example of such a test program: a non-trivial manipulation of an object, by defining an object and modifying its properties via property accessors. In the first line of the program, we define an array by assigning to the variable `a` an array literal with elisions. The right part evaluate as an array object value that does not have a mapping to values in positions 0, 2, 3, and 5. The indices 1, 4, and 6 are respectively mapped to the string `"one"`, the string `"two"`, and to the number `3`. Then, in the second line, we create a new ordinary object and assign it to the variable `o`. We define two properties in

```
1  let a = [,"one",,,"two",,3];
2  let o = new Object();
3  o.name = "o";
4  o["vec"] = a;
5  (o["vec"])[0] = "modified in line 5";
6  o.vec[0] == a[0]
```

Figure 4.15: Example of JS program

the following lines. The property `"name"` is mapped to the string value `"o"`. We write the access to the property in dot notation style, where `o.name` where o.name is a way to access the property `"name"`. As the objects are partial mappings from property names to values, when a property is not yet defined, inspecting it would result in an `undefined` result. Then, the property `"vec"` maps to a reference to the array object `a`, instantiated in the second line of the program. We modify the value of the property `"vec"` at index 0 by assigning it the string `"modified in line 5"`. Note that we used different kinds of property accessor to test most of them. By modifying the value of the `"vec"` property, we add a property 0 to the referenced array object `a`. The property 0 is a mapping to the string value. The evaluation of the comparison expression in line 6 result in `true`.

We wrote different programs, similar in size, to test non-trivial features of the semantics of expressions and statements we implemented. The programs results are correct.

4.3 Related Work

We first motivate why we chose JavaScript(JS) to evaluate the mechanization of a language using Skeletal Semantics. The three defining features of JavaScript in this regard are the following. First, it is complex, hence a good candidate to see if our solution scales up. Second, it has a precise specification, called ECMAScript (ES), and a large suite of test cases, hence we do not have to guess what its semantics is. And third, it has been mechanized in several frameworks [56, 25, 6, 59, 58, 57], which facilitates the comparison to other approaches.

There has been a previous experience to mechanize JS in Coq, called JSCert [6]. Defining a semantics in a proof assistant is the most direct way of mechanizing it.

It has a major drawback, however. If the design choices are not correct, one cannot manipulate the mechanization to modify it, and one must instead redo it using different choices. JSCert is a pretty-big-step [15] Coq definition of ECMAScript 5.1. It consists of an inductive definition, a recursive definition, and a correctness proof showing they match. The goal of the inductive definition is to prove properties of the language and of JS programs, whereas the recursive definition can provide an OCaml interpreter using Coq's extraction mechanisms. The mechanization is fairly close to the specification for people who can read Coq code, and the test suite can be run using the extracted interpreter. JSCert has two flaws, unfortunately. It is difficult to maintain, as one has to update two formalizations and a proof. In addition, and most importantly, JSCert uses Coq's induction to represent the recursive evaluation of the language. This shallow embedding results in a definition of a semantics as an inductive definition consisting of about 1000 rules, which is too large to prove properties of the language. Some authors of JSCert built upon this work to derive systematic and reusable definition of semantics [9, 8].

An other approach to the formalization of JavaScript is to translate it to a much simpler language, whose semantics is less complex. This is done for instance in the setting of [25], where JavaScript programs are translated to a simple language called λ_{JS}. This work is based on ECMAScript versions 3 and 5. The authors empirically show that the translation of JS programs is correct by running the JS test suite. As this formalization is not textually close to the specification, it is difficult to assess whether it actually captures the language beyond running the tests. In addition, the translation has not been updated since 2015, in particular it does not include the many changes to the core data structures and algorithms from ECMAScript 6 (or ECMAScript 2015). Attempts to formalize λ_{JS} in Coq[27] uncovered several issues with the desugaring process that were not witnessed by testing. It did not lead to a formal mechanization of the language.

Alternatively, one may use an existing framework designed to describe and manipulate semantics. $\mathbb{K}_{\text{JS}}$ [56] is a complete mechanization of ECMAScript 5.1 in the $\mathbb{K}$ framework. It provides an executable interpreter of JS directly from the semantics with no additional effort. This framework is not suitable to analyse the language itself as it only provides tools to reason about the execution of programs. In addition, there is no evidence that the mechanization can be easily maintained: although JS has significantly evolved since ES 5.1 (the specification has more than doubled in size), the

27. `https://github.com/tilk/LambdaCert`

mechanization has not been updated. Indeed, works such as JSExplain [16], a JS interpreter written in OCaml, show that updating an ES formalization is far from being a trivial issue. The power of the $\mathbb{K}$ framework comes at the cost of additional complexity in the description of languages in it, which hinders maintainability and closeness to the specification.

Part III

Distributed Semantics

Chapter 5

An Executable Semantics for Distributed IoT Applications

Contents

« Mi muovo! So parlare! Cammino! »
- Pinocchio, C. Collodi

Introduction

In Chapter 4, we applied the approach presented in Chapter 3 to describe a Skeletal Semantics of JavaScript's specification, handling the effects of the language by carefully choosing monads and combining them, hiding most of the information in the meta-environment of Skel. Nevertheless, despite JavaScript's intrinsic complexity, the model is mainly deterministic. Indeed, except for the `for-in` construct, which loops on a set of ECMAScript's values, the rest of the language constructs behave sequentially.

Essentially, until now, we dealt with formalizing programming languages, reasonably complex to represent but *deterministic*.

In this chapter, we investigate Skel from another perspective. We want to explore and study a non-deterministic model with more behavioral complexity than the languages we described before. In particular, we write a Skeletal Semantics of a model representing the behavior of distributed systems. The model represents interactions among the server tier, the client tier, and smart devices, describing the Internet of Things (IoT). Indeed, connected devices interact with the physical world, possibly changing its state. The goal is to derive an interpreter from this model's Skeletal Semantics and study its usability.

To this end, we present first a small-step semantics of a distributed model we call `WEBI`. We also set design choices for writing an interpreter-oriented version of `WEBI` in Skel. This model is complex in its evolution, producing possibly infinite execution traces. Hence, the interpreter we generate using `necroml` does not scale, possibly never producing an output on non-trivial programs. Thus, to execute and study it, we tame its non-deterministic behavior. We define constraints and a scheduling policy for making it executable. The scheduling policy we describe exploits interleaving invariants between the tiers' code execution. These invariants permit factorizing different equivalent traces by permuting trace elements when possible. We say that two different traces are equivalent if they manipulate the world's state in the same way—a world manipulation results from an interaction between devices and the real world. For example, turning on a connected heater in the living room changes the temperature read by a thermometer, making this new digital value representative of the world state. Then, the order of these actions matters. Opening a window before or after activating the heater changes the state of the world. We define these invariants to preserve the ordering between sensitive world manipulations resulting from the tiers' code execution.

This chapter has a double interest:

1. Show that Skeletal Semantics and Skel easily capture non-deterministic models,
2. Pave the way to further formal analyses on distributed IoT applications.

We structure the chapter in the following way. Section 5.1 provides context and discusses the previous work. Section 5.2.1 introduces the model by discussing its components, and Section 5.2.2 introduces both `WEBI`'s small-step and Skel semantics. Moreover, we present a design for the evaluation function that exploits Skel features for writ-

ing non-deterministic semantics. Afterward, we show how to derive a non-deterministic concrete interpretation, using a particular interpretation monad and a way to obtain collecting semantics. Section 5.2.3 shows a simple example of an application that does not scale on this model, forcing us to tame the non-deterministic behavior. The motivation comes with the writing of the evaluation function, which is ineffective with the collecting behavior, essential for moving towards applying analysis techniques. Hence, Section 5.3.1 proposes a scheduler for `WEBI`, which bounds the evaluation function's non-determinism. We conclude with Sections 5.3.2 and 5.3.3, respectively presenting an equivalence theorem for the scheduled semantics alongside the sketch of its proof.

Contribution

- a Skel formalization of `WEBI`.
- a scheduler for constraining the non-determinism of the model.
- a proof sketch of equivalence of the scheduler semantics.
- an OCaml interpreter of the model.

5.1 Context

Several questions related to security and safety spawned as soon as the interest of private companies grew toward IoT devices and applications. Many IoT products are available on the market, making such distributed systems part of our daily life. Every device connected to the internet can be attacked, making academic studies on IoT relevant. In this context, formal methods and the security scientific community are interested in finding ways to detect and prevent security flaws. To summarize, we can factorize the scientific question into the following: *How can an attacker exploit these systems to gain information or control?*

Many works define desugaring of the semantics of existing languages for analyzing IoT applications. For example, SaiNT [13], Soteria [14], and ProvThings [74], among many others, perform static analysis techniques on these distributed programs that rely on real-world languages, such as Groovy [2]. These works aim to detect information

about security flaws concerning this software category. Their analysis techniques are standard, comprehending mostly taint analysis and some abstract interpretation and symbolic execution. Nevertheless, the focus of these works is more directed toward security. Indeed the languages they formalize fit the security purpose, not aiming to be either a correct or a complete implementation of the semantics.
Moreover, if one wants to use the same techniques of analysis in other languages means redefining the tools without guarantees that the approach can scale. Indeed, modeling distributed systems is a problem far from being trivial. The complexity of distributed systems comes when each actor/tier (server, client, database) uses a different language, and programs run on different execution contexts. The cost of analyzing these programs individually and putting together intermediate results is computationally expansive. Additionally, some attacks are not detectable on these models, such as the cyberphysical ones. These non-digital and indirect attacks trigger a cascade of events in the physical world that leads to some malfunction.

Formal models to capture the semantics of distributed systems exists, and they are concerned more with client/server models. For example, the semantics of hop.js [11] elegantly describes the semantics of the language. Still, this is a semantic definition of a language with no other purpose than mathematically explaining the programming language's behavior and its programs. Indeed, not defining other tiers, such as the device one, makes it impossible to use the model to represent some non-deterministic interleavings that may provoke cyberphysical attacks.

This chapter proposes a novel semantics called `WEBI`, implemented in Skel. This model is an extension of the essence of [11], and it is designed to capture the behavior of IoT programs executing in a unique execution context. This choice alleviates some issues related to the analysis of distributed systems. What we describe in the manuscript relates more to semantics than analysis, discussing issues and limits. Nevertheless, we show that we can exploit Skel to define collecting semantics, paving the way for future studies toward program analysis. The model is parametric on the tier languages; we instantiate it with toy languages. However, this does not limit its use, as the model can be instantiated with some JavaScript extensions [34, 33], describing the semantics of multi-tier languages such as hop.js.

5.2 WEBI: A Formal Semantics to IoT Applications

This section presents a small-step semantics of WEBI, designed for capturing interactions between different tiers in the context of IoT applications. The actors we consider are:

- *Web services* that provide services to the clients.
- *Web clients* that request services.
- *Smart objects*, which are objects connected to the web. We consider devices with sensors—electronic components that transform physical data into digital data– and actuators—electronic components that transform digital data into physical data.

5.2.1 The WEBI Configuration

A WEBI configuration is a 7-uple containing all the information about the actors in the model, namely clients, services, and devices. We write the configuration as conf, and its extensively represented as:

$$\{\texttt{WebServices}, \mathcal{L}, \mathcal{I}, HM, SS, CC, IC\}$$

In the following paragraphs, we explain all the WEBI configuration in detail. In Table 5.1, we summarize the notation.

WebServices. The WebServices component is a function

$$\texttt{WebServices} : url \rightarrow (\texttt{service}, \texttt{hostname})$$

mapping *url*s, which we conceive as a generic way to encode the unique address of a web service, to the actual service and hostname. The hostname is an identifier for the server hosting the service. We let the WebServices to be implicit in the WEBI configuration, because it is never changed by the WEBI rules.

World's Events Record $\mathcal{L}$. The $\mathcal{L}$ component is a record of the *world's events*. With world events, we mean statements or data representing some natural condition of an

Symbol	Description	Skel type
wo	world oracle	$(\mathcal{L}, \mathcal{E}) \rightarrow (pe, t)$
$\mathcal{E}$	set of possible events that can occur in the world	`list<event>`
conf	a webi configuration $\langle \mathcal{L}, \mathcal{I}, HM, SS, CC, IC \rangle$	`webiConfiguration`
`WebServices`	mapping between urls and related services	`url→((var,sr_stmt),hostName)`
$\mathcal{L}$	list of located physical events: either actuator-produced events a, d, t or sensored-events pe, d, t	`logEvents`
$\mathcal{I}$	set of initializers	`initializers`
CC	set of web clients (cc)	`webClientConfigurations`
SS	set of web server configurations (ss)	`runningServiceConfigurations`
IC	set of smart objectconfigurations (ic)	`deviceConfigurations`
HM	hosts memory: $h \rightarrow \mu_h$	`hostMemories`
Π	sequence of `WEBI` rule applications	`trace`
h	name of a host	`hostName`
μ_h	memory of the host of name h	`hostMemory`
u	url	`url`
v_a	serialized value	`serializedValue`
v_c	client value ($v_a \subseteq v_c$)	`cl_val`
v_s	server value ($v_a \subseteq v_s$)	`sr_val`
P	program (client or server), note that a value may be a program	`cl_stmt`, `sr_stmt`
cc	a running web client configuration in CC ($\langle cc, B, T \rangle^{(j,u)}$)	`webClientConfigurations`
ss	a running service state in SS ($ss^{(h,j),i}$)	`runningServiceConfiguration`
sc	a running service configuration (ss)	`runningServiceState`
B	set of call-backs	`callbacks`
T	set of thunks	`thunks`
pe	located physical event	`physicalEvent`
l	event location	`location`
t	absolute time	`time`
d	device ID	`deviceID`

Table 5.1: `WEBI` summary of notation

object or an environment in the real world, or an action changing its state. Examples could be the statements: “The temperature in the kitchen at 8:00 p.m. is 18°” or “Light turned on at 10:30 p.m. in the bedroom”. We can represent this record as a list of all the actuations and sensor’s detections performed by devices.

On the one hand, an actuation is an action performed by a device through its actuators. This kind of action forcibly changes the state of the world. Indeed, consider the heater example presented in the introduction. The action that turns on the device is an actuation, and the effect (change of the world’s state) is the living room’s temperature increase. We see this as a digital action transformed into a concrete one.

On the other hand, a device that detects physical data from world does not change it; it “queries” it. This operation can be seen as the inverse of an actuation, transforming

concrete data, the world's natural state, into its digital representation.
The $\mathcal{L}$ record, intuitively, is empty before running the `WEBI` model. We represent it with the usual empty list notation, []. Each time a rule needs to record an operation on this list, we write it as $op :: \mathcal{L}$, where op is either an actuation or a device sensor result. If op is an actuation, we write it as (a, d, t), where a is the action, d is the device identifier, and t is the time this actuation happened. Otherwise, we write a sensor detection as (pe, t), where pe is the physical event that the device sensed from the world. An example of pe could be: "The living rooms's temperature is 18° Celsius". Concerning the time, we do not provide a specific mapping between real-world time and its digital representation.

Initializers $\mathcal{I}$. The $\mathcal{I}$ element of the configuration is a collection of the initial clients' calls; we call them *initializers*. We write the model initializer as $((j, u), v_a)$, where j is a *unique* identifier for the client, u is the URL that the *booting client* j is calling, and v_a is a value put as a parameter of the initial call. To run the model, we need this set not to be empty in the initial `WEBI` configuration. Without booting clients, also called initializers, the model will not evolve. We denote the empty $\mathcal{I}$ as $\emptyset$, like the set notation. Indeed, $\mathcal{I}$ is a set.

Host Memories HM. The third element of the `WEBI`'s `conf` is HM, a mapping between host-names h and host-memories μ_h, which are memories shared among different web services executing on the same host. The letter μ refers to the actual data structure acting as memory. The subscript h is the name of the host. We write mappings in the usual functional way: $h \mapsto \mu_h$. We do not specify if the host-memory mappings are total or partial.

Running Services SS. The SS component is the set of *running services*. We write the element of this set as ss, calling it *running web service state*. The running web service state can be either a value or some running program, and we write them respectively as $\langle sv \rangle$ and $\langle sc \rangle$. The first is a *final running service configuration*, a couple (v_s, μ_s), where v_s is a value in the server language and μ_s is the service's memory. The second represents a *running service configuration*. It is a couple (P_s, μ_s), where P_s is a program and μ_s is the local memory of the service. Both the $\langle v_s \rangle$ and $\langle sc \rangle$ have unique identifiers $((h, j), i)$, written as a superscript of the running web service state ss. This identifier is three-folded:

- h is the host-name on which the service is running.
- j is a reference to the client who called the service.
- i is the identifier of the client's callback that will handle the service return.

Web Clients CC. The CC component is the set of web clients. We write CC's elements as $\langle \mathsf{boot} \rangle^{(j,u)}$ for the *booting* clients, and $\langle cc, B, T \rangle^{(j,u)}$ for the *non-booting* ones. We use two combined identifiers as the client's ID: the j is the client's unique identifier, and u is the URL the client called when booting. The special keyword `boot` in the booting client configuration is a special client's callback waiting for the service's response to initialize the client. Then, the non-booting client is a client that has been already initialized. It is a web client configuration holding three pieces of information: the client program state cc, and two sets B and T. The cc can either be a couple composed of a client value v_c and a memory μ_c, or a couple (P_c, μ_c), where P_c is a client program still executing. Then, B and T are the callback's set—containing all the client's (j, u) call handlers still waiting for their related service response–and the thunk's set—the callbacks which have already received a result from their related services. Intuitively, we define a *callback* as a program $b = \lambda x.P$, where x is a free variable waiting to be bound to the result of the related service. Then, given v_a, the result of the client call, a thunk t of the callback b is the substitution of v_a in all x's occurrences in $\lambda x.P$. We write the thunks as $(\lambda x.P, v_a)$. The purpose of not substituting directly the value into the program is to not give the idea of substituting and executing the program P. The substitution can also be written as the η-expansion of the program, resulting in the following: $(\lambda x.\lambda z.P)\ v_a$. The variable z is a free variable in P of type unit. To run the program, then one must only apply $()$ to $\lambda z.P[x \to v_a]$. To resume, the variable x is bound to the value returned by the service, and P is not run.
Each callback and thunk is identified by i. We remark that if we pick a callback b^i from the B of the client (j, u), and imagine a function resolving the URLs and returning the host-name (i.e., `solveURL(u) = h`), then we can produce the related service identifier: $((h, j), i)$.

Device Configurations IC. Finally, the IC component is the set of *device configurations*. A device configuration $\langle \mu_d, S, A, \mathsf{perm} \rangle^d$ is defined by μ_d the device memory, S a set sensor, a set of actuators A, and a set of permissions perm—for interacting

with other tiers, namely the server-tier. Each element of IC is signed by d, a unique identifier for the device.

5.2.2 Semantics

The section presents the formal semantics of the `WEBI` distributed system model. We present the rules in a small-step style alongside their Skel implementation. Regarding Skel, we present some design choices to make the semantics more precise and suitable for building an interpreter.

Regarding the non-deterministic evaluation function in Skel, we know that the implementation would come pretty straightforward, because we exploit the non-determinism intrinsic in the Skel's `branch` construct. Intuitively, the interpreter will be instantiated with a pseudo-random interpretation monad, which shuffles the branch's list before interpreting its head.

To present the semantics, we structure the following subsections in this way. First of all, we briefly introduce the `WEBI` transition relation in Section 5.2.2.1. Then, we present the rules and the instantiation of the model. Indeed, in Section 5.2.2.2, we define a set of client-driven rules. The ruleset represents either the client evaluation progression or describes an interaction between the client and server tier. Then, Section 5.2.2.3 proposes rules for handling service-driven rules, and Section 5.2.2.4 describes the device-driven ones, handling interactions between services, devices, and the world's state. We will not discuss the client, the server and the device transition relation, because it would mean to introduce a semantics for each one of the tier's languages. We will only describe some of the main features that the client[1] and the server[2] language must have. In the example languages, we wrote variants of the WHILE language presented in Chapter 1.

Finally, Section 5.2.2.5 presents the non-deterministic evaluation function and the OCaml interpreter instantiation.

Implementation Choices in Skel. In the Skel implementation we partition the SS and the CC—written as `_SS_` and `_CC_` in Skel- `WEBI` configuration components. We

1. `https://gitlab.inria.fr/skeletons/webi-in-skel/-/blob/refactor/semantics/webi/clientLanguage.sk`
2. `https://gitlab.inria.fr/skeletons/webi-in-skel/-/blob/refactor/semantics/webi/serverLanguage.sk`

partition the functionally, in order to pick elements from the correct partition efficiently. The choice we made is to be sure, if any, to make the rules pick the correct web client and/or web service, avoiding failure.

We define `_CC_` as a set of four subsets:

- `_CC_r` is a subset of `_CC_` (or CC) containing all the web clients that can step or that are booting. In math-mode, we write it as CC_r.
- `_CC_c` is a subset of `_CC_` (or CC) containing all the web clients that call. In math-mode, we write it as CC_c.
- `_CC_t` is a subset of `_CC_` (or CC) containing all the web clients that with a non-empty set B or T. In math-mode, we write it as CC_t.
- `_CC_e` is a subset of `_CC_` (or CC) containing all the web clients that have B and T empty. In math-mode, we write it as CC_e. These web clients become inactive in `WEBI`, meaning their `webClientConfiguration` will never change again.

We define the following invariant for the subsets in Definition 1.

Definition 1 *It holds that $CC_r \cup CC_c \cup CC_t \cup CC_e = CC$, and the intersection between each couple of the subsets of CC is the empty set.*

Moreover, by design, we know that in case of no active web clients in the `WEBI` configuration, we can terminate the model's execution. We show this in Definition 2

Definition 2 *If all the clients are in the subset CC_e, no client is active in `WEBI`.*
Then, `WEBI` can terminate.

Symmetrically, we define subsets of `_SS_` as:

- `_SS_r` is a subset of `_SS_` (or SS) containing all the running web services that can step. In math-mode, we write it as SS_r.
- `_SS_a` is a subset of `_SS_` (or SS) containing all the running web services that have to issue an actuation request to a device. In math-mode, we write it as SS_a.
- `_SS_g` is a subset of `_SS_` (or SS) containing all the running web services that have to issue a reading request to a device. In math-mode, we write it as SS_g.

- `_SS_e` is a subset of `_SS_` (or SS) containing all the running web services that finished computing. In math-mode, we write it as SS_e.

We define the following invariant for the subsets of `_SS_` in Definition 3.

Definition 3 *It holds that $SS_r \cup SS_a \cup SS_g \cup SS_e = CC$, and the intersection between each couple of the subsets of SS is the empty set.*

Moreover, we argue about services being active. The web services run if a client called them. We present this property in Definition 4.

Definition 4 *If no client is active, then no service is running.*

5.2.2.1 `WEBI` Transition Relation

We say that a `WEBI`'s transition relation relates configurations presented in Section 5.2.1; we write it in the following way.

$$\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow \mathsf{conf}'$$

Note that the `WEBI` transition relation has two parameters that model the transition. The first parameter is the *world oracle* wo, and the second one is $\mathcal{E}$, a list of all the possible physical events that might happen in the world. Regarding the wo, it is a function modeling the world. This function provides a new event happening in a particular context. The parameters of this function are the world's events log $\mathcal{L}$ and the $\mathcal{E}$, producing a physical event pe happening at a specific time t.

5.2.2.2 Client-Driven Rules

`ServiceInit`. The first rule we present is the `ServiceInit`. This rule instantiates a booting client and its related booting service picking an element from the set $\mathcal{I}$. We present it in Figure 5.1.

Given an initializer, the rule resolves the URL via the function `WebServices`, returning a couple $(\mathsf{serviceinit}, h)$. The serviceinit is the *service initializer*, and h is the hostname of the host providing the service. The function application $\mathsf{serviceinit}(v_a)$ returns a service configuration sc, a program encoded in the service language and a fresh program memory, with a local variable x bound to v_a. The `WEBI` configuration is updated

ServiceInit

$$\frac{\mathcal{I} = ((j,u),v_a) \cup \mathcal{I}' \qquad \texttt{WebServices}(u) = (\mathsf{serviceinit}, h) \qquad sc = \texttt{serviceinit}(v_a) \qquad CC' = CC \cup \{\langle \mathsf{boot} \rangle^{(j,u)}\} \qquad SS' = SS \cup \{sc^{((h,j),0)}\}}{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{\mathcal{I}, CC, SS\} \rightsquigarrow \mathsf{conf}\{\mathcal{I}', CC', SS'\}}$$

Figure 5.1: The `ServiceInit` rule. A client is initialized. The booting call of this client generate a running web service.

accordingly. We write it as $\mathsf{conf}\{\mathcal{I}', CC', SS'\}$, meaning that the configuration conf modifies only the $\mathcal{I}$, the CC, and the SS components.

We want to remark the id signing the booting service $(((h,j),0))$. The i identifier is explicitly set to 0. By convention, we set 0 as the callback identifier of the special booting client callback boot.

In Figure 5.2, we present the Skel formalization of the `ServiceInit` rule. The evaluation function takes the *world setting*, an alias for the pair $(\mathsf{wo}, \mathcal{E})$, a `WEBI` configuration conf, and returns a configuration. We design the Skel `WEBI` configuration as a record. We present the type below.

```
type webiConfiguration = (
     _L_ : logEvents,
     _I_ : initializers,
     _HM_ : hostMemories,
     _SS_ : runningServiceConfigurations,
     _CC_ : webClientConfigurations,
     _IC_ : deviceConfigurations)
```

The body of the Skel term defines the rule. Thus, we get an element from the *initializers* set. The `pick` function randomly chooses an element from $\mathcal{I}$. It returns a pair `(((u,v_a), j), _I_`, where the first element is an initializer, and the second is $\mathcal{I} \setminus ((j,u),v_a)$.
Then, we define the `webservices` function as a mapping `url -> (serviceInits,hostname)`. The `serviceInits` is a function that, given a value, returns a *service configuration*. Thus, similarly to the rule in Figure 5.1, we generate a service configuration `sc` and up-

```
type webClientConfiguration =
| Boot (identifier,url)
| Run ((webClient,callbacks,thunks)(identifier,url))

val eval_ServiceInit ((wo,_E_):worldSetting)
                     (conf:webiConfiguration):
                     webiConfiguration =
    let (((j,u),v), _I_') = pick<webClientBoot> conf._I_ in
    let (serviceinit,h) = webservices(u) in
    let sc = serviceinit(v) in
    let _CC_' = _CC_add conf._CC_ (Boot (j,u)) in
    let _SS_' = _SS_add conf._SS_ (sc,((h,j),zero)) in
    (conf <- (_I_ = _I_', _CC_ = _CC_', _SS_ = _SS_'))
```

Figure 5.2: `ServiceInit` Skel rule.

date the `WEBI` configuration `conf`. To update a Skel record we use the following syntax:

```
(recordName <- (field1 = newValue))
```

On the left-hand side of the arrow, we have the record name (a variable named `recordName`), and on the right-hand side, we assign a new value (`newValue`) to a field (`field1`) only in the case we are actually updating it. The rest of the fields keep their old value. In Figure 5.2, we update only the fields related to $\mathcal{I}$, SS, and CC. Note that on top of the figure, we present extensively the `webClientConfiguration`. The `Boot` constructor represents booting clients and the `Run` constructor represents the running ones.

`ClientStep`. The `ClientStep` rule describes what happens in the `WEBI conf` when a client performs a single evaluation step on its program. The client transition is parametrized by C, the client language semantics. We write this transition relation as $\rightsquigarrow_c$, relating a web-client configuration cc to another. To apply this rule, the client must be running.

In Figure 5.3, we present the small-step `ClientStep` rule. In Figure 5.4, we present the skeletal semantics of this rule. In line 10, we pick one of the running web clients from `_CC_r`, a partition of the `webClientConfigurations` CC containing only the clients

ClientStep

$$\frac{CC = \{\langle cc, B, T\rangle^{(j,u)}\} \cup CC' \qquad cc = (P_c, \mu_c) \qquad cc \rightsquigarrow_C cc' \qquad CC'' = \{\langle cc', B, T\rangle^{(j,u)}\} \cup CC'}{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{CC\} \rightsquigarrow \mathsf{conf}\{CC''\}}$$

Figure 5.3: The `ClientStep` rule. A web client performs an evaluation step.

```
type clientStatus =
| Ok
| NotifyCall (url,cl_val,(var, cl_stmt))
type cl_out =
| CL_Value cl_val
| CL_STMT (cl_stmt,clientStatus)
val eval_ClientStep ((wo,_E_):worldSetting)
                    (conf:webiConfiguration):
                    webiConfiguration =
  let (Run (((CL_STMT (cp, OK),mu_c),_B_,_T_),(j,u)),_CC_r') =
                                pick<webClient> conf._CC_._CC_r in
  let cc' = eval_client cp mu_c in
  let _CC_'' = _CC_add (conf._CC_ <- (_CC_r = _CC_r'))
                       (Run ((cc',_B_,_T_),(j,u))) in
  (conf <- (_CC_ = _CC_''))
```

Figure 5.4: `ClientStep` Skel rule.

that have to step. Note that we edulcorate the semantics, by designing and using the constructed `type webClientConfiguration`—presented in Figure 5.1)-for respectively saying that a client is booting—$\langle \mathsf{boot}\rangle^{(j,u)}$–or running—$\langle cc, B, T\rangle^{(j,u)}$. The picked element signals the evaluation function whether the client can or cannot step. Note that on top of the figure we show the `type clientStatus`. In this rule we show, after the pattern matching, that the web client has a signal `Ok` stored in the `webClientConfiguration`. This constructor is a signal meaning that the web client can step. The type constructor `CL_STMT` is another signal, presented in the `type cl_out` definition, which signals that the client still has something to compute. All the elements of `_CC_r` are web clients that are programs (`CL_STMT`) that signal `Ok` to `WEBI`.

ClientCall

$$\frac{\begin{array}{ccc} CC = \{\langle cc, B, T\rangle^{(j,u)}\} \cup CC' & & \\ cc \rightsquigarrow_C^{u'?v_a,b^i} cc' & \mathsf{WebServices}(u') = (\mathsf{serviceinit}, h) & sc = \mathsf{serviceinit}(\mathsf{v_a}) \\ B' = B \cup \{b^i\} & CC'' = \{\langle cc', B', T\rangle^{(j,u)} \cup CC'\} & SS'' = \{sc^{((h,j),i)}\} \cup SS \end{array}}{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{CC, SS\} \rightsquigarrow \mathsf{conf}\{CC'', SS''\}}$$

Figure 5.5: The `ClientCall` rule. A client calls a web service.

```
1  val eval_ClientCall ((wo,_E_):worldSetting)
2                      (conf:webiConfiguration):
3                      webiConfiguration =
4      let (Run (((CL_STMT (cp,Notify (u',v_a,(b,i))),mu_c), _B_,_T_),
5           (j,u)),_CC_c') = pick<webClient> conf._CC_._CC_c in
6      let (serviceinit,h) = web services(u') in
7      let sc = serviceinit(v_a) in
8      let _B_' = _B_add _B_ (b,i) in
9      let _CC_'' = _CC_add (conf._CC_ <- (_CC_c = _CC_c'))
10         (Run (((CL_STMT (cp,OK),mu_c),_B_',_T_),(j,u))) in
11     let _SS_' = _SS_add conf._SS_ (sc,((h,j),i))) in
12     (conf <- (_CC_ = _CC_'', _SS_ = _SS_'))
```

Figure 5.6: `ClientCall` Skel rule.

In line 12 of Figure 5.4, the client's program performs an evaluation step, and in line 13, the `_CC_` set is updated by first updating `_CC_r` with the partition without the picked `webClient (j,u)` (`(conf._CC_ <- (_CC_r = _CC_r'))`), and finally adding the new `webClientConfiguration` in `_CC_`. Finally, in line 15, we update the `webiConfiguration`.

`ClientCall`. A client calls a service. The call occurs on a client step that sends a value v_a on a URL u while preparing a handler b, signed with a unique callback identifier i. On the server-side, a service is ready to respond to this URL. We show the rule in Figure 5.5. A web client is picked from the partition set `_CC_c`, containing all the clients performing a call. Note that all the calling clients have `NotifyCall` as `clientStatus`. We show this in lines 4 and 5 of Figure 5.6. Then, similarly to the `ServiceInit` rule,

Run

$$\frac{CC = \{\langle\langle v_c', \mu\rangle, B, \{(\lambda x.P, v_c)^i\} \cup T\rangle^{(j,u)}\} \cup CC' \quad CC'' = \{\langle\langle (P\{x \mapsto v_c\}, \mu\rangle, B, T\rangle^{(j,u)}\} \cup CC'}{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{CC\} \rightsquigarrow \mathsf{conf}\{CC''\}}$$

Figure 5.7: The `Run` rule. When a client finishes to compute, it picks a thunk and executes it.

lines 6 and 7 create an instance of the service referenced by the URL `u`. In the Skel specification, we decided to evaluate with a client step the call construct of the client, producing a `webClientConfiguration` with the adequate `clientStatus`. This makes the implementation not represent the client relation $\rightsquigarrow_C^{u'?v_a,b^i}$. Afterwards, we update the sets `_B_`, `_CC_`, and `_SS_` by adding the new callback b^i, the client with the `OK` status, and the new running web service signed by `((h,j),i)`. This web service is unique. Finally, in the last line, we return the updated `WEBI` configuration.

`Run`. A client finished computing, returning a value. If any, one of its thunks is picked, is put as the current client's code, and run. The client's memory stays unchanged. We show the rule in Figure 5.7. The `Run` only applies if a client has finished executing—returning a value–and has a thunk to be run.

In Figure 5.8, we show the skeletal semantics of this rule. We first pick a WebClient from `_CC_t`, if it exists. The latter is a partition of `_CC_` containing every client that finished computing and has a thunk ready to be executed or, in case of none, at least a callback. Then, we pick one of the thunks of the web client; we substitute the parameter of the callback program with the value stored in the thunk. Note that we discard the value returned by the client. Imagine that the value is the serialization of an HTML page. In that case, the result is local and meaningless regarding this orchestrating semantics. Finally, the set `_CC_` is updated, returning the new `WEBI` configuration. The implementation can fail in line 6, as the elements of `_CC_t` can have an empty set `_T_`. In this case, no client is ready to execute a thunk, as their callback still did not receive a value due to service execution.

```
1  val eval_Run ((wo,_E_):worldSetting)
2               (conf:webiConfiguration):
3               webiConfiguration =
4      let (Run (((CL_Value _, mu_c),_B_,_T_),(j,u)),_CC_t') =
5                                   pick<webClient> conf._CC_._CC_t in
6      let ((((x,_P_),v_c),_T_'),i) = pick<thunk> _T_ in
7      let _P_' = subst _P_ x v_c in
8      let _CC_'' = _CC_add (conf._CC_ <- (_CC_t = _CC_t'))
9           (Run (((CL_STMT (_P_', OK),mu_c),_B_,_T_'),(j,u)))  in
10     (conf <- (_CC_ = _CC_''))
```

Figure 5.8: `Run` Skel rule.

5.2.2.3 Service-Driven Rules

`ServiceStep`. One of the running services makes an execution step; its host memory updates accordingly. We present the small-step rule in Figure 5.9. In the implementation depicted in Figure 5.10, we first use the usual `pick<a>` function to extract an element from the partition set of interests—`_SS_r`. In case of success, the element picked from `_SS_r` is a client that runs normally its code. Indeed, symmetrically with the client's partitioning, the `_SS_r` is a partition of `_SS_` containing all the running services that can step. The service picked is one that evaluates code different from an actuation or a `get` from a device. These two operations are designed as blocking operations in the Skel implementation. For being able to do a `ServiceStep`, we design a `type serviceStatus` stored in `sc` (lines 10 and 11) that must be either `Ok`, similarly to the client status type, or `GetDone v_s`. We allow the web services signed by `GetDone v_s` to step because this constructor means that the `get` operation finished succesfully returning a result.

`ServiceStep`

$$\frac{SS = \{\langle sc\rangle^{((h,j),i)}\} \cup SS' \qquad \langle sc, HM(h)\rangle \rightsquigarrow_S \langle sc', \mu'_h\rangle \qquad HM' = HM[h \mapsto \mu'_h] \qquad SS'' = \{\langle sc'\rangle^{((h,j),i)}\} \cup SS'}{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{SS, HM\} \rightsquigarrow \mathsf{conf}\{SS'', HM'\}}$$

Figure 5.9: The `ServiceStep` rule. A service performs an evaluation step.

The service step in this case acts for communicating to the service the value `v_s`. We show the `serviceStatus` on top of the listing—`type serviceStatus`.

The `_SS_add` function adds the `runningServiceState` to the correct partition of `_SS_`. In line 12 of Figure 5.10, we get the host memory `mu_h` and perform a single server evaluation step. The parameters are the server computation `sc`, the local memory `mu_ls`, and the host memory `mu_h`. The result of the evaluation step is a triple with a new server computation `sc'`, local memory `mu_ls'`, and host memory `mu_h'`. Finally, the host memories map `_HM_` and the set `_SS_` are updated, returning the new `WEBI` configuration.

`RetServiceBoot` and `RetService`. One server reaches a state where its code returns an immediate serialized value while the associated client is already running and has some handler ready to take a response; the handler becomes a thunk after interpretation of the serialized value response by the server. We remove this service from the `WEBI` configuration. Two rules depend on the client's state, whether a booting client or a running one. First, we discuss the rule for booting clients, depicted in Figure 5.11. Once the service related to a booting client converges to a value, the special callback

```
1  type serviceStatus =
2  | Ok
3  | ActCall (sr_actuator, sr_deviceID, sr_permission)
4  | GetCall (sr_deviceID, sr_permission)
5  | GetDone sr_val
6
7  val eval_ServiceStep ((wo,_E_):worldSetting)
8                       (conf:webiConfiguration):
9                       webiConfiguration =
10     let (((sc,mu_ls),((h,j),i)),_SS_r') =
11         pick<runningServiceState> conf._SS_._SS_r in
12     let mu_h = _HM_r(conf._HM_,h) in
13     let ((sc',mu_h'),mu_ls') = eval_server sc mu_h mu_ls in
14     let _HM_' = _HM_w(conf._HM_,h,mu_h') in
15     let _SS_'' = _SS_add (conf._SS_ <- (_SS_r = _SS_r'))
16                                  ((sc',mu_ls'),((h,j),i)) in
17     (conf <- (_HM_ = _HM_',_SS_ = _SS_''))
```

Figure 5.10: `ServerStep` Skel rule.

RetServiceBoot

$$\frac{SS = \{\langle v_s \rangle^{((h,j),0)}\} \cup SS' \qquad CC = \{\langle \mathsf{boot} \rangle^{(j,u)}\} \cup CC' \qquad \mathsf{gencc}(v_s) = cc \qquad CC'' = CC' \cup \{\langle cc, \emptyset, \emptyset \rangle^{(j,u)}\}}{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{SS, CC\} \rightsquigarrow \mathsf{conf}\{SS', CC''\}}$$

Figure 5.11: The `RetServiceBoot` rule. A service related to a booting client finishes to compute. It returns the result to the client.

boot becomes the actual value returned by the service. We assume that a service always returns code to a booting client. The function `gencc` takes the service execution result as a parameter and generates a client configuration. The rule is abstract enough to hide information about a possible client's local memory.

In Figure 5.12, we show the semantics in Skel. In this case, we choose to be more precise in defining how to generate the first running client signed by `j` and `u`. First, we pick a service that has finished executing from the partition set `_SS_e` (lines 4 and 5). This subset contains all the services that have finished computing.
In line 6 the booting client signed by `j` is extracted from `_CC_`. The function `pick_by_j` is a specialized function that looks for a client by identifier. Afterwards, we use the parsing function `gencc` to transform the server value result into a client program. This emulates the fact that in a web application, the server sends an HTML page which will be parsed and executed—for example, the JavaScript parts-on the client side—browser. The client program generation happens in line 7; the web client configuration is completed by a fresh new client memory and two empty sets, one of the callbacks and the other of thunks. Finally, we update the set `_CC_` and return the final `WEBI` configuration. Note that we discard the service `(h,j)`. This rule fails if a service returns a value to a booting client.

If the client is a running web client, things go slightly differently. First, services return values to the clients (pure values). Note that the URL `u` signing the client never changes. We present the rule for running clients in Figure 5.13. In this case, the value v_s returned by the service $((h, j), i)$ is transformed into a client value v_c. It is associated with the callback b^i of the client j waiting for that result. The callback becomes a thunk, as it can now be associated with a client value, represented as a pair $(b, v_c)^i$. The Skel code goes straightforward with what we did for the booting case but is consistent

```
val eval_RetServiceBoot ((wo,_E_):worldSetting)
                        (conf:webiConfiguration):
                        webiConfiguration =
    let (((SR_Value vs,_),((h,j),zero)),_SS_e') =
                pick<runningServiceState> conf._SS_._SS_e in
    let (Boot (j,u),_CC_r') = pick_by_j conf._CC_._CC_r j in
    let CL_STMT cc = gencc(vs) in
    let _CC_'' = _CC_add (conf._CC_ <- (_CC_r = _CC_r'))
                         (Run (((CL_STMT cc,cl_init_state),
                                empty_set<callback>,
                                empty_set<thunk>),(j,u))) in
    (conf <- (_SS_ = (conf._SS_ <- (_SS_e = _SS_e')), _CC_ = _CC_''))
```

Figure 5.12: `RetServiceBoot` Skel rule.

RetService

$$\frac{SS = \{\langle v_s \rangle^{((h,j),i)}\} \cup SS' \qquad CC = \{\langle cc, B \cup \{b^i\}, T \rangle^{(j,u)}\} \cup CC' \\ \mathsf{gencc}(v_s) = v_c \qquad CC'' = CC' \cup \{\langle cc, B, T \cup \{(b, v_c)^i\} \rangle^{(j,u)}\}}{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{SS, CC\} \rightsquigarrow \mathsf{conf}\{SS', CC''\}}$$

Figure 5.13: The `RetService` rule. A service realted to a running client finishes to compute. It returns the result to the client.

with the `RetService` rule. We show it in Figure 5.14. This rule fails in case the service returns some code to the client.

5.2.2.4 Device-driven Rules

`DeviceSensor`. A device or a group of devices detects a physical event from the world. This physical event is stored in the world log $\mathcal{L}$. To model the sensor detection, we use the world oracle wo. It takes the world log $\mathcal{L}$ and the set of possible events $\mathcal{E}$, and it forecasts a physical event pe at a particular time t'. The notion of time we use is discrete, providing an order to the different logs in $\mathcal{L}$. The new physical event and the time it happens are stored in the log. We present the small-step rule in Figure 5.15, and provide the Skel implementation in Figure 5.16. In Skel, we first produce a physical

```
val eval_RetService ((wo,_E_):worldSetting)
                    (conf:webiConfiguration):
                    webiConfiguration =
    let (((SR_Value vs,_),((h,j),i)),_SS_e') =
                pick<runningServiceState> conf._SS_._SS_e in
    let (Run ((cc,_B_,_T_),(u,_)),_CC_') =
                _CC_pick_by_j conf._CC_ j in
    let ((b,i),_B_') = _B_get_by_i _B_ i in
    let CL_Value vc = gencc(vs) in
    let _T_' = _T_add _T_ ((b,vc), i) in
    let _CC_'' = _CC_add _CC_' (Run ((cc,_B_',_T_'),(u,cid))) in
    (conf <- (_SS_ = (conf._SS_ <- (_SS_e = _SS_e')), _CC_ = _CC_''))
```

Figure 5.14: `RetService` Skel rule.

DeviceSensor

$$\frac{\texttt{last-time}(\mathcal{L}) \leq t' \qquad \mathsf{wo}(\mathcal{L}, \mathcal{E}, t') = pe \qquad IC \rightsquigarrow_D^{\mathsf{sens}(pe)} IC' \qquad \mathcal{L}' = (pe, t') :: \mathcal{L}}{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{\mathcal{L}, IC\} \rightsquigarrow \mathsf{conf}\{\mathcal{L}', IC'\}}$$

Figure 5.15: The `DeviceSensor` rule. A sensor, or a group of sensors detects a physical event.

event `pe` at a specific time `t`. We could use the Skel existential construct to produce a time `t'` that satisfies $\texttt{last-time}(\mathcal{L}) \leq t'$, meaning that `t'` is the present. We decided to rely on the implementation of the world oracle to produce this parameter. Then we perform a transition on the device semantics, add the new physical event to the list $\mathcal{L}$, and update the `WEBI` configuration. We remark that the `eval_device` evaluation function is a parameter of the model. Hence we decide to leave it dependent on the implementation for now. In the current language example, this transition inspects the devices' memory returning the physical event resulting from these readings.

We remark that this rule always applies and is independent of service calls such as `get` or `act`. Notice that the difference between sensing and getting a value is that the first only logs the resulting physical event in the $\mathcal{L}$ list, while the result of a `get` is returned to a web service.

```
val eval_DeviceSensor ((wo,_E_):worldSetting)
                      (conf:webiConfiguration):
                      webiConfiguration =
    (*we do not use last-time*)
    let (pe,t') = world.wo(conf._L_, world._E_) in
    let (_IC_',_) = eval_device conf._IC_  (DV_Sens pe) in
    let _L_' = list_add<logEvent>(conf._L_,(Log_PE (pe,t'))) in
    (conf <- (_L_ = _L_',_IC_ = _IC_'))
```

Figure 5.16: `DeviceSensor` Skel rule.

DeviceActuator

$$\frac{\begin{array}{c} SS = \{\langle sc\rangle^{((h,j),i)}\} \cup SS' \\ \langle sc, HM(h)\rangle \rightsquigarrow_S^{\mathsf{act}(a,d,p)} \langle sc', \mu_h'\rangle \qquad HM' = HM[h \mapsto \mu_h'] \qquad IC \rightsquigarrow_D^{\mathsf{act}(a,d,p)} IC' \\ SS'' = \{\langle sc'\rangle^{((h,j),i)}\} \cup SS' \qquad \texttt{last-time}(\mathcal{L}) \leq t' \qquad \mathcal{L}' = (a,d,t') :: \mathcal{L} \end{array}}{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{\mathcal{L}, SS, HM, IC\} \rightsquigarrow \mathsf{conf}\{\mathcal{L}', SS'', HM', IC'\}}$$

Figure 5.17: The `DeviceActuator` rule. A service issues an actuation order to a device. The device performs the actuation.

`DeviceActuator`. A service orders a device d to do an action a with permission token p. To act, the device of name d should have the action a in its allowed actions set. Moreover, the permission p token should hold. While acting, the device's state is updated, and the action is recorded in the world state. The relation $IC \rightsquigarrow_D^{a,d,p} IC'$ does precisely that. The actual/concrete actuation cannot happen if one of these requirements does not hold. In Figure 5.18, we show the Skel implementation of the small-step rule presented in Figure 5.17. In the lastfigure, we implement the premises of the rule straightforwardly. First, we pick a service from the `_SS_a` partition set, which contains web services requesting an actuation to a device. Then we apply the device transition relation. Finally, we update the `WEBI`'s sets `_SS_`, produce a discrete actuation time `t'`, and add the log to the world log `_L_`. We update the `WEBI` configuration accordingly. Note that we use a similar approach to the one used for the `ClientCall`, not performing the service transition but changing the `serviceStatus` from `ActCall` to `Ok`. We design the `act` as a blocking operation in the service language, and changing

```
1  val eval_DeviceActuator ((wo,_E_):worldSetting)
2                          (conf:webiConfiguration):
3                          webiConfiguration =
4      let (((SR_STMT (sc,ActCall (a,d,p)),mu_ls),((h, j),i)),_SS_a') =
5                          pick<runningServiceState> conf._SS_._SS_a in
6      let _IC_' = eval_device conf._IC_ (DV_Act (a,d,p)) in
7      let _SS_'' = _SS_add (conf._SS_ <- (sr_act = _SS_act))
8                           ((SR_STMT (sc,SR_OK),mu_ls),((h, j),i)) in
9      let t' = last_time(conf._L_) in
10     let _L_' = list_add<logEvent>(conf._L_,(Log_Act (a,d,t'))) in
11     (conf <- (_L_ = _L_', _SS_ = _SS_'', _IC_ = _IC_'))
```

Figure 5.18: `DeviceActuator` Skel rule.

DeviceReading

$$\frac{SS = \{\langle sc\rangle^{((h,j),i)}\} \cup SS' \qquad \langle sc, HM(h)\rangle \rightsquigarrow_S^{\mathsf{get}(d,p,v_a)} \langle sc', \mu_h'\rangle \qquad IC \rightsquigarrow_D^{\mathsf{get}(d,p,v_a)} IC' \qquad HM' = HM[h \mapsto \mu_h'] \qquad SS'' = \{\langle sc'\rangle^{((h,j),i)}\} \cup SS'}{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{SS, HM, IC\} \rightsquigarrow \mathsf{conf}\{SS'', HM', IC'\}}$$

Figure 5.19: The `DeviceReading` rule. A service issues an reading order to a device. The device responds, returning a serialized value.

the status allows the service to step again.

`DeviceReading`. A web service requests a value stored in a device's memory. Similarly to the `DeviceActuator` rule, the service must have the correct permission p to perform this operation. The device reading happens if all these requirements are satisfied. We show the small-step rule in Figure 5.19.

In the implementation, presented in Figure 5.20, after picking a service requesting a `get`, an element of the partition set `_SS_g` (status `GetCall`), we perform a device transition issuing the order of reading the memory of the device `d`. The result of the step in line 6 is a pair `(_IC_',VSome v_a)`, where `IC'` is the new `deviceConfigurations` set, and `v_a` is the serialization of the read value. We do not make the service step, and again we manipulate the computation status from `GetCall` to `GetDone v_a`, the constructor designed to notify the service of the result of the device reading.

```
val eval_DeviceReading ((wo,_E_):worldSetting)
                        (conf:webiConfiguration):
                        webiConfiguration =
    let (((SR_STMT (sc,GetCall (d,p)),mu_ls),((h, j),i)),_SS_g') =
                        pick<runningServiceState> conf._SS_._SS_g in
    let (_IC_', v_a) = eval_device conf._IC_ (DV_Get (d,p)) in
    let _SS_'' = _SS_add (conf._SS_ <- (_SS_g = _SS_g'))
         ((SR_STMT (sc,GetDone v_a),mu_ls),((h, j),i)) in
    (conf <- (_SS_ = _SS_'', _IC_ = _IC_'))
```

Figure 5.20: `DeviceReading` Skel rule.

5.2.2.5 The Evaluation Function

In the introduction of the section we mentioned how easy would have been to implement a non-deterministic scheduling policy for `WEBI`. Withouth digging into the client, server and device language instantiation, in this subsection we present how to straightforwardly implement `WEBI` in Skel.

While introducing Skel in Chapter 2, we discussed about the interpretation monad and the way we deeply embedd the Skeletal Semantics in OCaml via the `necroml` tool. Hence, because of the definition of Skel [8], we can provide different interpretations to our specification language, delagating the bahaviour to the interpreter monad. We are interested in two behaviors. The first one is to have a monad that defines a non-deterministic (actually pseudo-randomic) semantics to the `branch` construct. Each time the `branch` is evaluated we want that the list of branches shuffles. In this way we do not know a priori which is the head of the branch list. Then, we are interested to exploit the collecting semantics behavior of the Skeletal semantics, collecting all the possible successful executions. The latter is a first technique that paves the way to formal analysis.

In Figure 5.21 we present the pseudo-random interpreter monad. From the figure we see the `Rand` module expecting another interpreter monad definition as a parameter. For example, if we provide the identity monad `ID` presented in Figure 2.5, the behavior of the latter monad would be changed only when dealing with branches, not evaluating anymore sequentially the list of branches. In the following snippet we show the actual instantiation of this monad.

```
let shuffle l =
        let () = Random.self_init () in
        let lrand = Stdlib.List.map (fun c ->
                                    (Random.bits (), c)) l in
        Stdlib.List.sort compare lrand |> Stdlib.List.map snd

module Rand (M: MONAD) = struct
        include M
        let branch l =
                branch (shuffle l)
end
```

Figure 5.21: The pseudo-random interpretation monad

```
open Necromonads
module IDRand = Necromonads.Rand(Necromonads.ID)
```

The file `Necromonads.ml`[3] is the one containing some predefined interpretation monads provided with the `necrom` tool. To instantiate the collecting semantics behavior, we have a list monad which collects all the possible valid execution paths. We show this in Figure 5.22.

To write the actual evaluation function of `WEBI` in Skel, we just need to write a branch, each case having one of the rule's evaluation function. We present its structure in Figure 5.23. The code presented is pretty trivial. The `terminateWEBI`, is a function that exploits the aforementioned termination conjecture, stopping the `WEBI` evaluation as soon as no client is active. Otherwise, the model would work forever, ending a enumerable number of infinite traces, because of the `DeviceSensor` rule. The latter always apply producing a physical event added to the world log list $\mathcal{L}$. Concerning the interest we have towards this model, we think that whatever happens after the termination conjecture is not of our interest.

3. https://gitlab.inria.fr/skeletons/necro-ml/-/blob/master/necromonads.ml

```
module List = struct
        type 'a t = 'a list
        let ret x = [x]
        let branch l =
            Stdlib.List.concat (Stdlib.List.map (fun f -> f ()) l)
        let fail _ = []
        let rec union_two l = function
        | [] -> l
        | a :: q -> (if Stdlib.List.mem a l then (fun x -> x)
                     else (Stdlib.List.cons a)) (union_two l q)
        let rec union = function
        | [] -> []
        | a :: q -> union_two a (union q)
        let bind l f = union (Stdlib.List.map f l)
        let apply f x = f x
        let extract x =
                begin match x with
                | a :: _ -> a
                | [] -> failwith "No result"
                end
end
```

Figure 5.22: The List interpretation monad

Remark

The non-deterministic concrete interpretation uses the `IDRand` module.[a]

a. https://gitlab.inria.fr/skeletons/webi-in-skel/-/tree/master/semantics/webi

Remark

We exploit the termination conjecture to stop executing the model. If there are no clients and services manipulating the state of the world, then the physical events produced by the `DeviceReading` would be always the same. We deal with finite traces.

```
val eval_WEBI ((wo,_E_):worldSetting)
              (conf:webiConfiguration):
              webiConfiguration =
 branch let T = terminateWEBI conf in conf
 or     let F = terminateWEBI conf in
        let conf' =
            branch eval_ServiceInit (wo,_E_) conf
            or     eval_ClientStep (wo,_E_) conf
            or     eval_ClientCall (wo,_E_) conf
            or     eval_Run (wo,_E_) conf
            or     eval_ServiceStep (wo,_E_) conf
            or     eval_RetServiceBoot (wo,_E_) conf
            or     eval_RetService (wo,_E_) conf
            or     eval_DeviceSensor (wo,_E_) conf
            or     eval_DeviceActuator (wo,_E_) conf
            or     eval_DeviceReading (wo,_E_) conf
            end in
        eval_WEBI (wo,_E_) conf'
 end
```

Figure 5.23: `ServiceInit` Skel rule

5.2.3 Example: The Cost of Non-Determinism

This section presents a simple example of how our model executes. We want to initialize `WEBI` with one client, two services, and three devices that can interact.To initialize the model, we need to provide semantics in Skel for both the client and server language. As mentioned and linked in the introduction, these languages are variants of the `While` language presented in Chapter 1.

The client language is a `While` language extended with a call statement for calling web services. The server language is more complex than the client's. For the sake of simplicity, we describe the language as an extension of a `While` language with an `act` and `get` statements, and a special expression $\sim$. The `act` and the `get` are statements to interact with devices. The tilde expression is of the form `~(p)`, where `p` is a program written in the client language syntax. The program `p` may contain special variables—called $-variables. These variables bind variable names to values in the server memory. The whole `~(program)` expression produces a server value—in the server semantics-with $-variables occurrences resolved. For the client, this server value is a program to

be. The Skel semantics of these languages must be included on top of the `WEBI` source, using the keyword `include`. Thus, it is simple to write the `WEBI` semantics parametrized by the tiers languages. A limitation is that all the interpreters generated by the `necroml` tool need to use the same interpretation monad because of the `bind` function. Indeed, to bind different computations produced by the three interpreters means to extrapolate different notions of pure values and pass them to the continuations. [4]

5.2.3.1 Initial `WEBI` Configuration Setting

To provide an initial setting to the model, we introduce in the following paragraphs how we define the two services, the client, the definition of the three devices, and the definition of the two `WEBI` parameters: the world oracle `wo` and the set of the possible events `_E_`. Afterward, we show how this model executes, discussing its strengths and limitations.

Services. We define two services, one for handling the kitchen's window—opening or closing it- depending on the temperature in the room, and the other for turning on and off a connected oven in the same room. For example, a simple interaction of the oven with the kitchen's environment is to heat the room. We show the code of the two services, respectively, in Figures 5.24 and 5.25.

4. A concrete example is binding the resulting computation of the `ID` monad with the `List` one. Binding a pure value—an integer, for example- is different from binding a list of results—which is the pure value type for the latter monad.

```
(x,
 i = 0;
 get(t,("Thermometer","kitchen"),"A");
 while (i <= x) {
   if (t > 26) {
     act("open",("Window","kitchen"),"B");
   } else {
     act("close",("Window","kitchen"),"B");
   };
   i = i + 1;
   get(t,("Thermometer","kitchen"),"A");
 };
 return ~(return $t)
)
```

Figure 5.24: The `windowManager` service definition.

```
(_,
 act("on",("Oven","kitchen"),"C");
 return ~(return true)
)
```

Figure 5.25: The `turnOnOven` service definition.

```
(c,
 i = 0;
 while (true) {
   get(presence, (("presenceSensor", "appt256"),''A''))
   if (presence = "present") {
     act("on",("switch", "appt256"),"B");
     act("unlock",("door", "appt256"),"C");
   } else {
     act("off",("switch", "appt256"),"B");
     act("lock",("door", "appt256"),"C");
     get(time,(("clock","appt256"),"D"));(*not sure*)
     if (time >= 800 & time <= 1800) then {
       act("sendSMS",("phone", "appt256"),"E")
       (*Extend Webi ACT with param*)
   }}})
```

We say that a service is defined by a couple `(var,sr_stmt)`, where `var` is a free variable in the program of `type sr_stmt`. The program is written according to the syntax of the server language. The `windowManager` service executes for a number x of iterations, given as a parameter by the caller. Each iteration of the service gets the temperature from a connected thermometer. Then, if the temperature is higher than $26°$C, it opens. The program returns the final temperature got from the thermometer. Note that `act` has as parameters: an action a, a device identifier d—i.e. `("Window","kitchen")`-, and a permission token p. The `turnOnOven` service only turns on the oven and returns `true`.

Each service is identified by an url and a hostname of the machine on which this service executes. We write them below.

```
let windowManagerUrl = "windowManager.com/kitchen"
let hostWindowManager = "HostA"
let turnOnOvenUrl = "turnOnOven.com/kitchen"
let hostTurnOnOven = "HostB"
```

Client. We define one client: a client wanting to use the `windowManager` service to handle the window behavior automatically. While instantiating the model, these clients are initializers of the `WEBI` configuration. The client is an initializer of the initial WEBI configuration. We show the definition of the client and the initializer set `_I_` instantiation in Figure 5.26. Note that a `client`'s browser call is written as `windowmanager.com/kitchen?x=1`, where `?x=1` is the way to pass the parameter via a url. This parameter represents the number of iterations the `windowManager` will perform. To define `_I_`, we simply add the client to a list, the concrete type to implement sets.

Devices. We define three devices: the window, the thermometer, and the oven. We show how we design the window, then the others follow. To simplify, we design the device semantics as a record containing the operations that it offers and the device

```
let client = (("adam","windowmanager.com/kitchen"),Some 1)
let _I_ = [client]
```

Figure 5.26: Definition of a client calling the `windowManager` and the `_I_` set.

```
type dv_step = ( device : deviceName,
                 _GET_ : option<deviceMemory -> dv_val>,
                 _ACT_ : option<deviceMemory -> actuator -> deviceMemory>,
                 _SENS_ : option<deviceMemory -> dv_val -> deviceMemory>)
```

Figure 5.27: Type defining the device's semantics.

```
val window_step : dv_step =
     (device = "window",
      _GET_ = (Some<deviceMemory -> dv_val>
                (λ mu_d:deviceMemory -> dv_mem_get(mu_d,"window_status"))),
      _ACT_ = (Some<deviceMemory -> actuator -> deviceMemory>
                (λ mu_d:deviceMemory -> λ a:actuator ->
                   branch let DV_T = str_eq(a,"window open") in
                          dv_mem_set(mu_d, window_status, (Bool T))
                   or     let DV_T = str_eq(a,"window close") in
                          dv_mem_set(mu_d, window_status, (Bool F))
                   end)),
      _SENS_ = None<deviceMemory -> dv_val -> deviceMemory>)
```

Figure 5.28: The window semantics. The code is written in a simplyfied version of Skel, which has, for example, strings.

name. Figure 5.27 shows the Skel type definition of the structure containing the device semantics. The three operations we consider are `get`, `act`, and `sens`. The three fields are optional, meaning that devices might have only some of them defined. For example, a thermometer does not act. We show the actual implementation of the semantics of the window in Figure 5.28. The window has two actuators—`window_open` and `window_close`- and we handle them in the `_ACT_` field of the figure. We simply say that if the actuation is `"window open"`, the memory is updated to accordingly opening it; otherwise, to close it. The device's memory is a mapping from variables to device values. We store in the memory only one variable called `window_status`, which stores the actual status of the window. Because the window has no sensors, we set the `_SENS_` field to `None`. We define the `_GET_` to read the memory and return a value. We create the device's instance `kitchen_window` in the following listing.

```
(*Semantics*)
val thermometer_step :dv_step =
    (device = "thermometer",
     _GET_ = (Some<deviceMemory -> dv_val>
               (λ mu_d:deviceMemory -> dv_mem_get(mu_d, "temp"))),
     _ACT_ = None<deviceMemory -> actuator -> deviceMemory>,
     _SENS_ = (Some<deviceMemory -> dv_val -> deviceMemory>
                (λ mu_d:deviceMemory -> λ v:dv_val ->
                   dv_mem_set(mu_d, "temp", v)))
    )
```

```
(*Instantiation*)
(state = thermometer_init_memory(),
 _S_ = ["temp"],
 _A_ = [],
 perm = "A",
 id = ("thermometer","kitchen"))
```

Figure 5.29: The thermometer semantics and instantiation.

```
  (state = window_init_memory(),
   _S_ = [],
   _A_ = ["window open","window close"],
   perm = "B",
   id = ("Window","kitchen"))
```

In detail, the term `window_init_memory` returns a fresh memory for the device, becoming the window's initial state. The state has only one binding, the name `"window_status"` bound to `"closed"`. The set of sensors `_S_` is empty, as none exists. The field `_A_` is a set of the permitted actuators for the device. The field perm contains the permission token, and the field `id` represents the device identifier d. The other device definitions follow the window's one. We show the thermometer in Figure 5.29, which features one sensor but no actuators. We define the semantics of the `_SENS_` operation in the semantics, updating the thermometer memory in the case of a `DeviceSensor`. The `_GET_` reads the device memory.

```
1  val wo ((_L_,_E_):(logEvents,list<event>) ): (physicalEvent,time) =
2      let t' = log_time l in
3      branch let T = query_oven_kitchen_on l in
4             (((("thermometer","kitchen"),Nat 28),t')
5      or     let F = query_oven_kitchen_on l in
6             (((("thermometer","kitchen"),Nat 24),t')
7      end
8
9  type event =
10 | PE physicalEvent
11 | ACT actuator
```

Figure 5.30: The world oracle.

World Oracle and Events. We define these parameters of `WEBI` in Figure 5.30. The world oracle we describe is a function that returns the temperature depending on the state of the oven. Indeed, the oracle function inspects the oven's status, detecting `"oven off"` and returning the temperature of $24°$C; otherwise, $28°$C. This simple oracle models the influence of the oven on the environment in which it is placed. Notice that `wo` is a parameter of the semantics. Hence, one can design a more complex oracle, for example, one that models the physical world more accurately.

The `_E_` parameter is a set of events possible in the world we describe. This means that only the events defined in the set `_E_` can happen.

We define an *event* as a physical event or an actuator. A physical event represents the action of detecting a specific state in the world, and in Skel is represented by the type constructor `PE`. For example, the detection result of a thermometer can be "At time `t`, the temperature of the kitchen is 28°C".
In Skel it is written as `PE ((("thermometer","kitchen"),Nat 28), t)`. An actuator's action is a modification of the state of a device. An example is "At time `t`, the kitchen's oven has been turned on". We represent it with the type constructor `ACT`, and we write the actuator action described before as `ACT (("turn_on",("thermometer","kitchen")),t)`. We show the `event` type in Figure 5.30.

Initializing client "adam"
→ ServiceInit
Execution of the windowManager
→ ServiceStep → ServiceStep → DeviceSensor → DeviceReading → ServiceStep → ServiceStep → ServiceStep → ServiceStep → DeviceSensor → DeviceReading → ServiceStep → ServiceStep → ServiceStep
Returning the service result to the webclient "adam"
→ RetServiceBoot
Evaluating the result of the booting callback
→ ClientStep

Figure 5.31: A sequence of rule applications showing an execution of the initial setting presented in the previous paragraphs.

5.2.3.2 Execution

We are now ready to describe how the model executes. Concretely, we execute the initial configuration using the interpreter generated by the `necroml` tool, and the interpretation monad used is the `RandID`. Because of the evaluation function defined in Figure 5.23, we can theoretically obtain finite and infinite traces—for example, an infinite trace that contains an infinite number of `DeviceSensor` applications. As we test on a real interpreter, the interpreter quickly runs out of memory with this kind of trace.

Execution 1. Figure 5.31 presents an execution trace of the initial setting we previously defined. The figure shows a sequence of rule applications that reproduce the execution. For simplicity, we improperly refer to the sequences as traces. Hence, we consider a client, two services, and three devices. While booting, the client calls the `windowManager` service to handle the kitchen window. The first rule applied to the initial `WEBI` configuration is the `ServiceInit`. The client is a booting client whose related service is created and ready to execute. The booting call parameter of the client is the number 1, meaning that the service will perform only one iteration of the `windowManager`. This value is bound in the service's memory. At this point of the execution, only two rules can apply. The `ServiceStep` and the `DeviceSensor` rules. The trace we are considering applies the latter rule before each `get` or `act` interaction in the service code. Hence, after applying a service step, bounding the variable `i` to `0`, the trace shows a `get`.
This changes the `serviceStatus` of the service to `GetCall (("Kitchen","window"),"B")`. As before, there are only two rules which

can be applied, the `DeviceSensor` and the `DeviceActuator`. The trace first does a `DeviceSensor`, adding to `_L_` the physical event produced by the world oracle—`(((“thermometer”,“kitchen”),Nat 24),t')`-and then applies a `DeviceActuator`. The temperature returned to the service is $24°$C. This physical event is stored in the memory of the thermometer, accordingly to the `_SENS_` rule presented in Figure 5.29. The physical event produced by the rule reflects the world's state: the oven is off. Afterward, the service keeps evaluating its code by applying the `ServiceStep` rule, evaluating the `while` and the `if` construct—with the related guards-. The second branch is evaluated, as the `while`'s condition is satisfied, but not the `if`'s. The case we are considering is when the temperature is $\leq 26°$C. Because of the guard not satisfied, the else branch of the `if` statement is executed. Finally, a series of service steps evaluate the increment of the variable `i`, and a device sensor happens again before the `get`. The condition of the `while` is not satisfied anymore, so a series of service steps happen, evaluating the rest of the code and the `return`. The server expression `~(return t)` evaluates to `(return (Nat 24))`. Afterward, the `RetServiceBoot` rule is applied. It instantiates the running client, putting the service-generated program as the client's current program. After evaluating the `return` via a `ClientStep`, the `WEBI` evaluation function executes the term `terminateWEBI`, which is satisfied, returning the final `WEBI` configuration.

Execution 2. Now let us consider a variant of the initial setting, where we add a second client with permission tokens for interacting with the client's services in the previous setting. Indeed, we want the new client to call the `turnOnOven` service. We show the changes done to the initializers set `_I_` in the following snippet.

```
let client1 = (("adam","windowmanager.com/kitchen"),Some 1)
let client2 = (("steve","turnonoven.com/kitchen"),None)
let _I_ = [client1,client2]
```

Execution traces `A` and `B` are finite, like trace `A` presented in Figure 5.31. Both the traces first do two `ServiceInit` to introduce the booting clients and running services into the `WEBI` configuration. Then, the evolution of the traces is different. The trace `A` first consumes the steps of the `turnOnOven`, by doing a service step, a device sensor, and a device actuation which turns on the oven. Afterward, it consumes, similarly to the previous execution example, the service `windowManager`. The result of executing

Rule Sequence A.
Initializing client "adam" and "steve"
→ ServiceInit → ServiceInit
Execution of the turnOnOven service
→ ServiceStep → DeviceSensor → DeviceActuator → ServiceStep → ServiceStep
Returning the turnOnOven result to the webclient "steve"
→ RetServiceBoot
Execution of the windowManager service
→ ServiceStep → ServiceStep → DeviceSensor → DeviceReading → ServiceStep → ServiceStep → ServiceStep → DeviceSensor → DeviceActuator → ServiceStep → ServiceStep → DeviceSensor → DeviceReading → ServiceStep → ServiceStep → ServiceStep
Returning the windowManager result to the webclient "adam"
→ RetServiceBoot
Evaluating the result of the booting callbacks
→ ClientStep → ClientStep

Rule Sequence B.
Initializing client "adam" and "steve"
→ ServiceInit → ServiceInit
Execution of the windowManager service
→ ServiceStep → ServiceStep → DeviceSensor → DeviceReading → ServiceStep → ServiceStep → ServiceStep → ServiceStep → DeviceSensor → DeviceReading → ServiceStep → ServiceStep → ServiceStep
Returning the windowManager result to the webclient "adam"
→ RetServiceBoot
Execution of the turnOnOven service
→ ServiceStep → DeviceSensor → DeviceActuator → ServiceStep → ServiceStep
Returning the turnOnOven result to the webclient "steve"
→ RetServiceBoot
Evaluating the result of the booting callbacks
→ ClientStep → ClientStep

Figure 5.32: Two sequences of rule applications showing executions of the new setting that considers two clients. The first one is a trace showing an interaction producing an interesting physical event, and the second one does not show this interaction.

the latter service, differently from before, is to open the window, as the oven has been turned on. This interaction between the two services and the related devices is indirect and implicit. Indeed, the window actuation happens because of the temperature of the kitchen raised by the oven, which is on. All the physical events and the actuations are logged in `_L_`, and as we can see in the figure, result in the state of the window as

"window open". Trace B consumes the services the other way around by consuming all the execution steps of the windowManager and then consuming the turnOnOven ones. This results in not observing the interaction between the oven and the window, the one produced by the execution trace A.

Discussion. By using the RandID monad, we can only observe one trace of a non-deterministic execution of WEBI. Suppose we are now interested to collect all the possible traces produced by an initial WEBI configuration. Collecting all possible traces can be useful, for example, for detecting malicious behaviors. In theory we can use the List monad to make the Skel WEBI execution collect all possible traces, but this approach does not scale because of non-determinism—the interpreter runs out of memory.

This motivates the following section, a scheduler for WEBI, in which we stipulate under which hypotheses we can build an interpreter that collects all traces for a given initial WEBI configuration.

5.3 A Scheduler for WEBI

We propose a scheduled version of the WEBI semantics, which tames the non-determinism inherent in the model by defining an efficient order in which the WEBI rules are applied.

Section 5.3.1 presents the scheduler's semantics by introducing the hypotheses under which the scheduler semantics is equivalent to the non-deterministic semantics given in Section 5.1. Section 5.3.2 formally describes some properties the scheduler exploits to be equivalent to the non-deterministic semantics. Section 5.3.2.3 presents an equivalence proof sketch of the scheduler semantics concerning the observations made in the non-deterministic semantics. Finally, we conclude with Section 5.3.3, showing the example in Section 5.2.3 executed by the scheduler.

5.3.1 Semantics

To build the scheduler, we first discuss the shape of the scheduler's evaluation function. We draw two diagrams in Figure 5.33, one reflecting the behavior of the non-deterministic evaluation function depicted in Figure 5.23—which spans horizontally- and the other showing the behavior of the scheduler's evaluation function—which spans

vertically. We want to bind the non-deterministic behavior of the evaluation function by providing an execution order to the rules that are not interdependent. Hence, commuting two rules' applications does not change the produced WEBI configuration. In the Skel code, if a branch fails, then another branch is picked for evaluation. Diagram B of the figure graphically presents the scheduler's evaluation function.

The scheduler's first iteration consumes all the ServiceInits upfront to introduce all the initializers of the configuration at once, emptying _I_. Afterward, the scheduler consumes all the ServiceSteps, elements of _SS_r. Once this partition is empty, we know that the configuration might have services that have finished computing—elements of _SS_e-, waiting to act—elements of _SS_a-, and waiting to get—elements of _SS_g. The scheduler first handles the finished services, repeatedly applying the RetServiceBoot, and creating the running web clients. Once the partition set of _SS_—_SS_e-is empty, the scheduler consumes elements of the partitions _SS_a and _SS_g. The current step makes the interaction between services and devices happen, if any. We design this step of the scheduler to act, get, or to skip this step. If the scheduler applies a device interaction rule, before applying it, performs a DeviceSensor to update the devices' memories with the physical events generated by the world oracle. In this way, the device memories reflect the status of the physical world. Otherwise, the scheduler moves to the next step. Note that the scheduler can also move to the client-driven rules if the partition sets _SS_a and _SS_g are empty. The next step consumes all the elements of the partition set _CC_r by repeatedly applying the ClientStep rule. Afterward, it consumes the client calls by repeatedly applying the ClientCall rule to the elements of _CC_c. Finally, if any, it runs the thunks of the web clients elements of _CC_t.
Once all the elements of _CC_t are processed, the scheduler completes its first iteration.

The successive iterations are slightly different. They do not do the ServiceInit step, which applies the RetService or the RetServiceBoot rule depending on which type of client the services are returning—booting or running.

5.3.1.1 Assumptions for the Scheduler

To tame the non-determinism, we must put constraints on the WEBI and tiers' semantics. The following hypotheses allow us to prove the equivalence between the scheduler and the WEBI non-deterministic semantics.

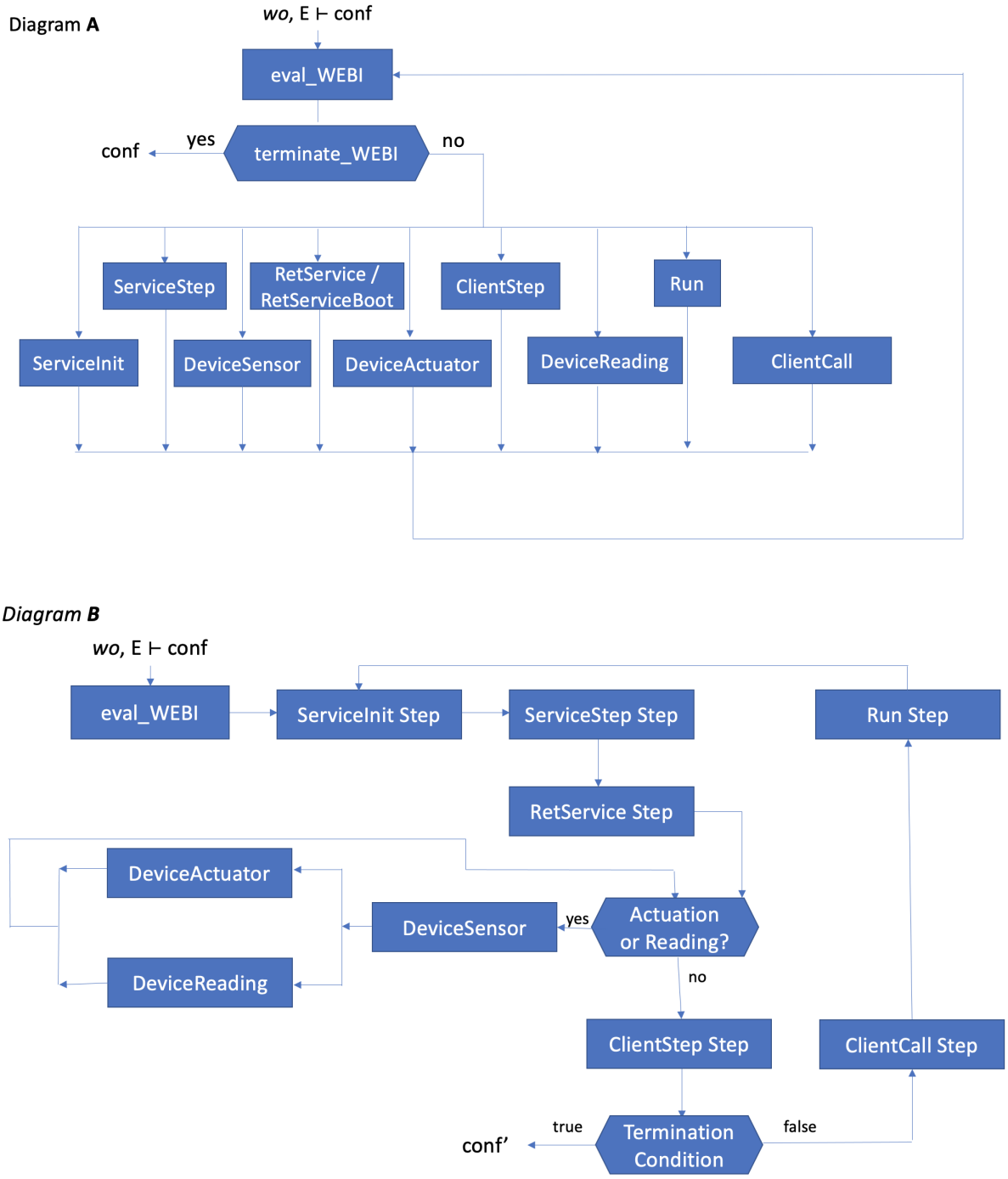

Figure 5.33: Two diagrams representing the evaluation functions. The diagram `A` depicts the non-deterministic evaluation function, and the diagram `B` depicts the scheduler's one.

Concerning the server and client language, we assume that they rely on a deterministic semantics. Moreover, the services are concurrent, modifying the host memory of the server they are executing on. We assume that a running service never touches the host memory. We present these constraints in Hypotheses 1, 2, and 3.

Hypothesis 1 *Given a web service configurations sc and a host memory μ_h, two different executions result in the same final configuration.*

$$\begin{aligned}&\forall \langle sc, \mu_h \rangle : \\ &\qquad (\langle sc, \mu_h \rangle \rightsquigarrow_S \langle sc', \mu'_h \rangle) \wedge \\ &(\langle sc, \mu_h \rangle \rightsquigarrow_S \langle sc'', \mu''_h \rangle) \Rightarrow (sc' = sc'') \wedge (\mu_h = \mu'_h)\end{aligned}$$

Hypothesis 2 *The client semantics is deterministic.*

$$\forall cc, cc', cc'' : \ (cc \rightsquigarrow_C cc') \wedge (cc \rightsquigarrow_C cc'') \Rightarrow (cc' = cc'')$$

Hypothesis 3 *Given a web service configurations sc and a host memory μ_h, executing one step of the semantics does not change the host memory.*

$$\begin{aligned}&\forall \langle sc, \mu_h \rangle : \\ &(\langle sc, \mu_h \rangle \rightsquigarrow_S \langle sc', \mu'_h \rangle) \Rightarrow \\ &\qquad (\mu_h = \mu'_h)\end{aligned}$$

To avoid ambiguity in choosing clients or services, we define two hypotheses concerning the uniqueness of the web services and web clients in a `WEBI` configuration. Every client and service running in a `WEBI` configuration is unique. We present these properties of the running model in Hypotheses 4 and 5.

Hypothesis 4 *Every running web service in a `WEBI` configuration is uniquely identified.*

$$\begin{aligned}&\forall \langle sc \rangle^{((h,j),i)}, \langle sc' \rangle^{((h',k),l)} \in SS : \\ &\quad h = h' \wedge j = k \wedge i = l \Rightarrow \\ &\qquad\qquad sc = sc'\end{aligned}$$

Hypothesis 5 *Every web client in a* `WEBI` *configuration is uniquely identified.*

$$\forall cc^{(j,u)}, cc'^{(j',u')} \in CC :$$
$$j = j' \wedge u = u' \Rightarrow$$
$$cc = cc'$$

5.3.1.2 Scheduler Configuration

We first introduce a set of labels to represent `WEBI` rules:

- $\mathbb{SS}$ is the label representing the `ServiceStep`. The label can hold indices (j, i), written as $\mathbb{SS}_{(j,i)}$, for making step the service identified by (j, i).
- $\mathbb{RS}$ is the label representing either the `RetService` or `RetServiceBoot`. The label can hold indices (j, i), written as $\mathbb{RS}_{(j,i)}$, for making return the service identified by (j, i).
- $\mathbb{CS}$ is the label representing the `ClientStep`. The label can hold an index j, written as $\mathbb{CS}_j$, for making step the client identified by j.
- $\mathbb{IN}$ is the label representing the `ServiceInit`. The label does not hold indices.
- $\mathbb{CC}_j$ is the label representing the `ClientCall`. The label can hold an index j, written as $\mathbb{CC}_j$, for making call the client identified by j.
- $\mathbb{RN}$ is the label representing the `Run`. The label can hold indices (j, i), written as $\mathbb{RN}_{(j,i)}$, to run the thunk i of the client identified by j.
- $\mathbb{DS}$ is the label representing the `DeviceSensor`. The label does not hold indices.
- $\mathbb{DA}$ is the label representing the `DeviceActuator`. The label can hold indices (j, i), written as $\mathbb{DA}_{(j,i)}$, for making a service (j, i) issuing an actuation.
- $\mathbb{DR}$ is the label representing the `DeviceReading`. The label can hold indices (j, i), written as $\mathbb{DR}_{(j,i)}$, for making a service (j, i) issuing a reading.
- $\mathbb{DV}$ is a generic device label. $\mathbb{DV} \in \{\mathbb{DS}, \mathbb{DA}, \mathbb{DR}\}$.

We add a special label `end` to signal that the `WEBI`'s scheduler has finished computing. We let $\mathbb{X}$ range over labels, *i.e.* $\mathbb{X} \in \{\mathbb{SS}, \mathbb{RS}, \mathbb{CS}, \mathbb{IN}, \mathbb{CC}, \mathbb{RN}, \mathbb{DS}, \mathbb{DA}, \mathbb{DR}, \mathbb{DV}, \texttt{end}\}$.

We define a scheduler transition as a relation between the scheduler's configurations. A scheduler configuration is an extended `WEBI` configuration enclosed in a scheduler's context $[\![]\!]$.

The `WEBI` configuration defined in Section 5.2.1 is extended with Π, written $\langle \mathcal{L}, \mathcal{I}, HM, SS, CC, IC, \Pi \rangle$, where Π is a finite list of labels. Intuitively, the sequence Π represents a `WEBI` execution trace and it will be useful for proving equivalence. We define in Definition 5 how an initial `WEBI` configuration is formed.

Definition 5 *A well-formed `WEBI` initial configuration, is a configuration* $\langle \mathcal{L}, \mathcal{I}, HM, SS, CC, IC, \Pi \rangle$*, where:*

- *Lists* $\mathcal{L}$ *and* Π *are empty lists.*
- *Sets* SS *and* CC *are empty sets.*
- *Set* $\mathcal{I}$ *is a non-empty set of clients that are booting.*
- *Mapping* HM *is a pre-instantiated map of host-memories* $\mathtt{hostname} \rightarrow \mathtt{hostmemory}$*.*
- *Set* IC *is a set of intial device configurations.*

The scheduler context holds three parameters, a label representing a rule name—meaning the current scheduling step executing-and two filtering of the trace Π—one containing device rule applications and the other having the order of the `Run` rule applications. The two filterings ensure the execution of the scheduler semantics to reproduce a final `WEBI` configuration equivalent to the one produced by `WEBI`.

We can write a scheduling context configuration as $[\![\mathsf{conf}]\!]_{\mathbb{X}}$, with the $\mathbb{X}$ denoting the generic scheduling step. For example, to write the scheduling step of the `ServiceStep` we write $[\![\mathsf{conf}]\!]_{\mathbb{SS}}$. Moreover, we use these labels to annotate the non-deterministic `WEBI` transition relation. Intuitively, $\rightsquigarrow_{\mathbb{X}}$ keeps the abstraction of the model, and $\rightsquigarrow_{\mathbb{SS}_{(j,i)}}$ states the application of a service step on a web client signed by (j, i).

Device and Thunk Filterings. We introduce two parameters which are lists of labels: $\mathfrak{L}$ for the device rule applications and $\mathfrak{T}$ for the `Run` rule applications. The purpose

Filtering Functions for Generating $\mathfrak{L}$

$$\pi_{\mathfrak{L}} = \lambda\Pi.\ \texttt{lfilter}\ \Pi\ (\lambda x.x\ \in\ \{\mathbb{DS}, \mathbb{DA}, \mathbb{DR}\})$$

Filtering Functions for Generating $\mathfrak{T}$

$$\pi_{\mathfrak{T}} = \lambda\Pi.\ \texttt{lfilter}\ \Pi\ (\lambda x.x\ \in\ \{\mathbb{RN}\})$$

Figure 5.34: Filtering functions on Π, where `lfilter` is the classic `filter` function on lists in functional programming.

is to drive the scheduler's execution and obtain a trace equivalent to the one produced by the non-deterministic execution of the model. Figure 5.34 presents the algorithms for generating the two execution guides, $\mathfrak{L}$ and $\mathfrak{T}$. We call the two filtering functions $\pi_{\mathfrak{L}}$ and $\pi_{\mathfrak{T}}$, which return a sub-trace given a trace Π.

Scheduler Transition Relation. We write the scheduler relation as $\rightsquigarrow_{\mathfrak{S}}$. Thus, the relation between scheduler's configurations has the following form:

$$\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}]\!]_{\mathbb{X}}^{(\mathfrak{L},\mathfrak{T})} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}']\!]_{\mathbb{X}'}^{(\mathfrak{L}',\mathfrak{T}')}$$

5.3.1.3 Small-Step Semantics

We present the scheduler's semantics by presenting each step in a paragraph. Given an initial `WEBI` configuration conf, an initial configuration for the scheduling is $[\![\mathsf{conf}]\!]_{\mathbb{IN}}^{(\mathfrak{L},\mathfrak{T})}$, for some $\mathfrak{L}$ and $\mathfrak{T}$.

Step 1, $\mathbb{IN}$. The first iteration of the scheduler starts evaluating all the initializers. This approach introduces all the booting clients and the related booting services. We show the semantic rules in Figure 5.35.

The rule `Init` performs a `ServiceInit` transition—Figure 5.1-in case the set $\mathcal{I}$ is not empty. Note that the left-hand side of the conclusions of the inference rule keeps the label $\mathbb{IN}$, stating that the scheduler step does not change.

Step 1

$$\textsc{Init}\quad \frac{\mathcal{I} \neq \emptyset \qquad \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{\mathcal{I}\} \rightsquigarrow_{\mathbb{IN}} \mathsf{conf}'}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{\mathcal{I}\}]\!]_{\mathbb{IN}}^{(\mathfrak{L},\mathfrak{T})} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}']\!]_{\mathbb{IN}}^{(\mathfrak{L},\mathfrak{T})}}$$

$$\textsc{InitDone}\quad \frac{\mathcal{I} = \emptyset}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{\mathcal{I}\}]\!]_{\mathbb{IN}}^{(\mathfrak{L},\mathfrak{T})} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{\mathcal{I}\}]\!]_{\mathbb{SS}}^{(\mathfrak{L},\mathfrak{T})}}$$

Figure 5.35: Semantic rules for Step 1

The rule `InitDone` makes the scheduler transit to the next step in case the set of initializers is empty. As we deal with a finite number initializing of clients, once the $\mathcal{I}$ set is emptied, it will not have new elements to initialize the next iteration. Step 1 of the scheduler is performed only at the first iteration. Note that this rule makes the scheduler transit to the second scheduler step by changing the label $\mathbb{IN}$ to $\mathbb{SS}$ in the context.

Step 2, $\mathbb{SS}$. The second step of the scheduling policy consumes all the possible `ServiceStep`s. This means that each time the rule is applied, an element of SS_r performs an execution step. The step finishes as soon as no more services can step. We present the semantics in Figure 5.36.

The `ServiceChosen` rule picks a service that can step and updates the scheduler. The rule label is set to context with $\mathbb{SS}_{(j,i)}$, meaning that the service (j, i) is the one chosen to step.

The `MultiServiceSteps` rule makes step the web service (j, i). If the service can do one more step, the label is left unchanged, updating only the configuration.

The `OneServiceStep` acts closely to the `MultiServiceStep` rule. The service performs one small evaluation step, and the rule label is set to $\mathbb{SS}$. This means that the service (j, i) cannot step again. Another service has to be chosen or, in case, transit to the next scheduling step.

The `ServicesDone` rule makes the scheduler transit to the next scheduling step. Hence, the SS_r partition set is empty, and the rule label changes to $\mathbb{RS}$.

Generally, the running service executing on a host are concurrent, as they share the host memory. In this preliminary study of `WEBI`, we decided to put constraints, saying that the running services never touch the host memory. This permits the `MultiServiceStep` to be correct. Moreover, we constrain the server language and consider only deterministic languages. These constraints are formally presented in Hy-

Step 2

$$\textsc{ServiceChosen}\quad \frac{SS = \{SS_r, SS_a, SS_g, SS_e\} \qquad SS_r \neq \emptyset \qquad \langle sc \rangle^{((h,j),i)} \in SS_r}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{SS}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{SS}_{(j,i)}}}$$

$$\textsc{MultiServiceSteps}\quad \frac{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{SS\} \rightsquigarrow_{\mathbb{SS}_{(j,i)}} \mathsf{conf}'\{SS'\} \qquad SS' = \{SS_r, SS_a, SS_g, SS_e\} \qquad \langle sc \rangle^{(((h,j),i)} \in SS_r}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{SS}_{(j,i)}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}'\{SS'\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{SS}_{(j,i)}}}$$

$$\textsc{OneServiceStep}\quad \frac{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{SS\} \rightsquigarrow_{\mathbb{SS}_{(j,i)}} \mathsf{conf}'\{SS'\} \qquad SS' = \{SS_r, SS_a, SS_g, SS_e\} \qquad \langle sc \rangle^{((h,j),i)} \notin SS_r}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{SS}_{(j,i)}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}'\{SS'\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{SS}}}$$

$$\textsc{ServicesDone}\quad \frac{SS = \{SS_r, SS_a, SS_g, SS_e\} \qquad SS_r = \emptyset}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{SS}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{RS}}}$$

Figure 5.36: Semantic rules for Step 2

potheses 3 and 1.

Step 3, $\mathbb{RS}$. The third step consumes all the elements of SS_e, and only `RetService` and `RetServiceBoot` apply.
We present the semantics in Figure 5.37.

The `RetServiceChosen` rule picks a service in SS_e. This service is related to a booting or a running client. Hence, one of the two applies. The `WEBI` configuration is updated accordingly.

The `RetServiceDone` rule makes the scheduler transit to the next step because of $SS_g = \emptyset$. The rule label changes to $\mathbb{DV}$. The `WEBI` configuration conf, $\mathfrak{L}$, and $\mathfrak{T}$ are immuted.

Step 3

$$\textsc{RetServiceChosen}\quad \frac{SS = \{SS_r, SS_a, SS_g, SS_e\} \quad SS_e \neq \emptyset \quad \langle sc \rangle^{((h,j),i)} \in SS_e \quad \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{SS\} \rightsquigarrow_{\mathbb{RS}_{(j,i)}} \mathsf{conf}'\{SS'\}}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{RS}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}'\{SS'\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{RS}}}$$

$$\textsc{RetServicesDone}\quad \frac{SS = \{SS_r, SS_a, SS_g, SS_e\} \quad SS_e = \emptyset}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{RS}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{DV}}}$$

Figure 5.37: Semantic rules for Step 3

Step 4, $\mathbb{DV}$. The fourth step of the scheduler consumes all the services interacting with devices. We are considering the $\mathbb{DS}$, $\mathbb{DA}$, and $\mathbb{DR}$ rules, and trace $\mathfrak{L}$ guides the application of these rules. We present the semantics in Figure 5.38.

The `DevSens` rule applies a `DeviceSensor` in case the head of the device-trace $\mathfrak{L}$ is $\mathbb{DS}$. The `WEBI` configuration `conf` is updated accordingly. Regarding the scheduler context, the new context contains the list $\mathfrak{L}$ without its head, and the rule label stays unchanged.

The `DevAct` rule, if the head of the $\mathfrak{L}$ is an actuator, applies the `DeviceActuation` rule to the `WEBI` configuration. The service requesting the `act` is (j, i), and must be an element of SS_a. The `WEBI` configuration and the $\mathfrak{L}$ are updated, and the rule label stays unchanged.

Similarly, the `DevRead` performs a `DeviceReading` requested by the service (j, i) in case the service belongs to SS_g. The head of the $\mathfrak{L}$ is removed, the configuration is updated, and the rule label does not change.

The `DevNoAct` and `DevNoRead` apply when respectively:

- The head of $\mathfrak{L}$ is an actuator request done by a service (j, i), but the service is not an element of SS_a.
- The head of $\mathfrak{L}$ is a read request done by a service (j, i), but the service is not an element of SS_g.

Step 4

$$\textsc{DevSens}\quad \frac{\mathfrak{L} = \mathbb{DS} :: \mathfrak{L}' \qquad \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow_{\mathbb{DS}} \mathsf{conf}'}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{DV}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}']\!]^{\mathfrak{L}',\mathfrak{T}}_{\mathbb{DV}}}$$

$$\textsc{DevAct}\quad \frac{\mathfrak{L} = \mathbb{DA}_{(j,i)} :: \mathfrak{L}' \qquad SS = \{SS_r, SS_a, SS_g, SS_e\} \qquad \langle sc \rangle^{((h,j),i)} \in SS_a \qquad \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{SS\} \rightsquigarrow_{\mathbb{DA}_{(j,i)}} \mathsf{conf}'\{SS'\}}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{DV}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}'\{SS'\}]\!]^{\mathfrak{L}',\mathfrak{T}}_{\mathbb{DV}}}$$

$$\textsc{DevRead}\quad \frac{\mathfrak{L} = \mathbb{DR}_{(j,i)} :: \mathfrak{L}' \qquad SS = \{SS_r, SS_a, SS_g, SS_e\} \qquad \langle sc \rangle^{((h,j),i)} \in SS_g \qquad \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{SS\} \rightsquigarrow_{\mathbb{DR}_{(j,i)}} \mathsf{conf}'\{SS'\}}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{\mathrm{SS}\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{DV}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}'\{SS'\}]\!]^{\mathfrak{L}',\mathfrak{T}}_{\mathbb{DV}}}$$

$$\textsc{DevNoAct}\quad \frac{\mathfrak{L} = \mathbb{DA}_{(j,i)} :: \mathfrak{L}' \qquad SS = \{SS_r, SS_a, SS_g, SS_e\} \qquad \langle sc \rangle^{((h,j),i)} \notin SS_a}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{DV}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CS}}}$$

$$\textsc{DevNoRead}\quad \frac{\mathfrak{L} = \mathbb{DR}_{(j,i)} :: \mathfrak{L}' \qquad SS = \{SS_r, SS_a, SS_g, SS_e\} \qquad \langle sc \rangle^{((h,j),i)} \notin SS_g}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{DV}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CS}}}$$

$$\textsc{DevStepDone}\quad \frac{SS = \{SS_r, SS_a, SS_g, SS_e\} \qquad \mathfrak{L} = [] \Rightarrow (SS_a = \emptyset \wedge SS_g = \emptyset)}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{DV}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{SS\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CS}}}$$

Figure 5.38: Semantic rules for Step 4

In this case, for keeping the device-rules application order, the rule makes the scheduler context transit to the next step to the scheduler. The rule label is set to $\mathbb{CS}$.

Finally, The `DevStepDone` applies when the list $\mathfrak{L}$ is empty. We know that if this trace is empty, then both the partition sets SS_a and SS_g are empty.
This rule application only changes the rule label to $\mathbb{CS}$.

Step 5, $\mathbb{CS}$ The fifth step handles the `ClientStep` application. We present the small-step semantics in Figure 5.39.

Step 5

$$\textsc{ClientChosen}\quad \frac{CC = \{CC_r, CC_c, CC_t, CC_e\} \qquad CC_r \neq \emptyset \qquad cc^{(j,u)} \in CC_r}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CS}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CS}_j}}$$

$$\textsc{MultiClientSteps}\quad \frac{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{CC\} \rightsquigarrow_{\mathbb{CS}_j} \mathsf{conf}'\{CC'\} \qquad CC' = \{CC_r, CC_c, CC_t, CC_e\} \qquad cc^{(j,u)} \in CC'}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CS}_j} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}'\{CC'\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CS}_j}}$$

$$\textsc{OneClientStep}\quad \frac{\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{CC\} \rightsquigarrow_{\mathbb{CS}_j} \mathsf{conf}'\{CC'\} \qquad CC' = \{CC_r, CC_c, CC_t, CC_e\} \qquad cc^{(j,u)} \notin CC_r}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CS}_j} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}']\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CC}}}$$

$$\textsc{ClientsDone}\quad \frac{CC = \{CC_r, CC_c, CC_t, CC_e\} \qquad CC_r = \emptyset \qquad |CC_e| \neq |CC|}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CS}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CC}}}$$

$$\textsc{WebiDone}\quad \frac{CC = \{CC_r, CC_c, CC_t, CC_e\} \qquad CC_r = \emptyset \qquad |CC_e| = |CC|}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{CC\}]\!]^{[],[]}_{\mathbb{CS}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{CC\}]\!]^{[],[]}_{\mathtt{end}}}$$

Figure 5.39: Semantic rules for Step 5

The `ClientChosen` rule picks a client to step from the partition set CC_r. If there is a client (j, u), the rule label is set to $\mathbb{CS}_j$.

The `MultiClientSteps` rule performs an execution step on the web client signed by (j, u). The client, still a member of CC_r in conf', is set again for evaluation. Hence, the rule label does not change, and the `WEBI` configuration is updated.

The `OneClientStep` rule performs an execution step on the web client (j, u). The client is no more in CC_r of conf'. Hence, the rule label is set back to $\mathbb{CC}$. The label change permits choosing another client to make an execution step, to go to the next scheduling step, or to terminate `WEBI`, according to Definition 2, page 128. The `WEBI` configuration is updated.

The rule `ClientsDone` applies if there are no more clients in the partition set CC_r.

Step 6

$$\textsc{CallChosen}\quad \frac{CC = \{CC_r, CC_c, CC_t, CC_e\} \qquad CC_c \neq \emptyset \qquad cc^{(j,u)} \in CC_c \qquad \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{CC\} \rightsquigarrow_{\mathbb{CC}_j} \mathsf{conf}'\{CC'\}}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CC}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}'\{CC'\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CC}}}$$

$$\textsc{CallsDone}\quad \frac{CC = \{CC_r, CC_c, CC_t, CC_e\} \qquad CC_c = \emptyset}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{CC}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{RN}}}$$

Figure 5.40: Semantic rules for Step 6

In this case, the rule label of the scheduler execution context is set to $\mathbb{CC}$. Note that the last premise is $|CC_e| \neq |CC|$, meaning the cardinality of the partition and the whole set CC differ. There are still clients that might call or have a thunk to run.

If the last premise is $|CC_e| = |CC|$, then the last rule applies, namely `WebiDone`. The rule label is set to `end`, meaning that the scheduler must stop executing, and conf becomes the final configuration.

Step 6, $\mathbb{CC}$ This scheduling step is concerned with applying, wherever possible, the CLIENTCALL rule. We provide the small-step rules in Figure 5.40. The `CallChoosen` rule picks a client j from the partition set CC_c.
The client performs a `ClientCall` step, returning an updated configuration. The scheduler context is updated with the new configuration conf′.

Otherwise, the `CallsDone` applies when the partition set CC_c is empty, meaning there are no more calls to evaluate this iteration. Hence, this rule updates the rule label to $\mathbb{RN}$ for going to the next step.

Step 7, $\mathbb{RN}$ In the last step of the scheduler, we present rules for handling the `Run` rule application. Similarly to the device's rules, the order in which thunks execute matters. For example, consider two thunks of a client, one that wants to call a service that opens the window and one that calls the one that closes it—the observations made on the final $\mathcal{L}$ of the final `WEBI` configuration change depending on the order of the thunk choice. Hence, this step is guided by $\mathfrak{T}$, the filtering of the non-deterministic trace describing

Step 7

$$\textsc{RunChosen}\quad \frac{\mathfrak{T} = (j,i) :: \mathfrak{T}' \qquad CC = \{CC_r, CC_c, CC_t, CC_e\} \qquad \langle cc, B, T\rangle^{(j,u)} \in CC_t \qquad t^i \in T \qquad \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{CC\} \rightsquigarrow_{\mathbb{RN}_{(j,i)}} \mathsf{conf}'\{CC'\}}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{RN}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}'\{CC'\}]\!]^{\mathfrak{L},\mathfrak{T}'}_{\mathbb{RN}}}$$

$$\textsc{RunNotFound}\quad \frac{\mathfrak{T} = (j,i) :: \mathfrak{T}' \qquad CC = \{CC_r, CC_c, CC_t, CC_e\} \qquad \langle cc, B, T\rangle^{(j,u)} \in CC_t \qquad t^i \notin T}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{RN}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{SS}}}$$

$$\textsc{RunDone}\quad \frac{CC = \{CC_r, CC_c, CC_t, CC_e\} \qquad \mathfrak{T} = [] \wedge CC_t = \emptyset}{\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{RN}} \rightsquigarrow_{\mathfrak{S}} [\![\mathsf{conf}\{CC\}]\!]^{\mathfrak{L},\mathfrak{T}}_{\mathbb{SS}}}$$

Figure 5.41: Semantic rules for Step 7

the order of the $\mathbb{RN}$ rule application.

The rule `RunChosen` applies the $\mathbb{RN}$ `WEBI` transition, executing the thunk (j,i), which is the head of the list $\mathfrak{T}$. A constraint is that the thunk i of the client j must exist. The rule returns the new `WEBI` configuration and updates the scheduler context with the $\mathfrak{T}$ deprived of its head.

Otherwise, the rule `RunNotFound` applies if the thunk does not belong to the client's set T. It means that the thunk is still a callback, or the client has not yet called the service (j,i). In this case, the rule label of the scheduler context changes to $\mathbb{SS}$. We keep the order of the $\mathbb{RN}$ applications. We remark that step 1 of the scheduler is executed only in the first iteration. Afterward, the scheduler does not consider further client initializations.

Finally, if the list $\mathfrak{T}$ is empty, no other client's thunk will ever be executed. The scheduler directly passes to the second step.

5.3.2 Equivalence of the Scheduler and the `WEBI` Semantics

The scheduler semantics we propose must achieve two goals:

- All the traces produced must be correct, by preserving the original behavior of the model.
- Reproduce all the traces that provide observations of sensitive events.

We define Theorem 1 saying that given an initial `WEBI` configuration setting conf, after a finite number of steps n, the model execution will produce a final configuration conf_n only if exists a number of finite steps on conf—$n+k$-that the scheduler can execute producing a final configuration $\mathsf{conf}_{(n+k)}$ equivalent to conf_n.
Intuitively, every `WEBI` non-deterministic execution produces the same observations as the ones resulting from a scheduler execution and vice versa.

Theorem 1 (Scheduler's Equivalence to `WEBI`) *Considering the constraints presented by Hypotheses 1, 2, 3, 4, and 5 given a well-formed initial `WEBI` configuration* conf*:*

$$\forall\ wo, \mathcal{E}, n, \Pi, \mathsf{conf}_n, \mathcal{L}, \mathcal{L}', \mathsf{conf}_n.$$
$$\exists\ k:$$
$$wo, \mathcal{E} \vdash \mathsf{conf}\{[], []\} \rightsquigarrow^{n} \mathsf{conf}_n\{\mathcal{L}, \Pi\}$$
$$\Leftrightarrow$$
$$(wo, \mathcal{E} \vdash [\![\mathsf{conf}]\!]_{\mathbb{IN}}^{(\pi_{\mathfrak{L}}(\Pi), \pi_{\mathfrak{T}}(\Pi))} \rightsquigarrow_{\mathfrak{S}}^{(n+k)} [\![\mathsf{conf}_{(n+k)}]\!]^{[],[]}) \ \wedge \ \mathsf{conf}_n = \mathsf{conf}_{(n+k)}$$

Considering the $\Leftarrow$ direction of the proof, the proof is trivial; the other direction is challenging. Indeed, we must prove that given it exists a transformation of every `WEBI` trace to an equivalent scheduler trace. It follows that, if the new trace is executed by the `WEBI` then it is executed also by the scheduler. The transformation of the trace is obtained via trace permutations. Hence, we define two series of lemmas:

- A series of lemmas representing an order relation among `WEBI` rule transitions—Section 5.3.2.1.
- A series of lemmas for constructing intervals that represent the scheduling steps—Section 5.3.2.2.

5.3.2.1 Commutation Lemmas

To achieve the goal of transforming a non-deterministic trace into a scheduler's one, we must define the commutation of consequent `WEBI` transition relations. We define

Generic form of the lemmas

$\forall \mathsf{conf}, \mathsf{conf}', \mathsf{conf}'', \mathbb{X}.$
$\mathsf{conf} \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'' \rightsquigarrow_{\text{Rule}} \mathsf{conf}' \ \wedge$
Constraints
$\Rightarrow$
$\exists \mathsf{conf}'''.$
$\mathsf{conf} \rightsquigarrow_{\text{Rule}} \mathsf{conf}''' \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'$

Rule	$\mathbb{X}$	Constraints
$\mathbb{IN}$	$\{\mathbb{IN}, \mathbb{SS}, \mathbb{RS}, \mathbb{CS}, \mathbb{CC}, \mathbb{RN}, \mathbb{DV}\}$	$\times$
$\mathbb{SS}_{(j,i)}$	$\{\mathbb{SS}, \mathbb{RS}_{(j',i')}, \mathbb{CS}, \mathbb{CC}, \mathbb{RN}, \mathbb{DS}, \mathbb{DA}_{(j',i')}, \mathbb{DR}_{(j',i')}\}$	$(j \neq j' \vee i \neq i') \wedge \mathsf{conf}\{CC\} \wedge (\langle cc, b^i \in B, T\rangle^{(j,u)} \in CC \vee \langle \mathsf{boot}\rangle^{(j,u)} \in CC)$
$\mathbb{RS}_{(j,i)}$	$\{\mathbb{RS}_{(j',i')}, \mathbb{CS}, \mathbb{CC}, \mathbb{RN}, \mathbb{DS}, \mathbb{DA}_{(j',i')}, \mathbb{DR}_{(j',i')}\}$	$(j \neq j' \vee i \neq i') \wedge \mathsf{conf}\{CC\} \wedge (\langle cc, b^i \in B, T\rangle^{(j,u)} \in CC \vee \langle \mathsf{boot}\rangle^{(j,u)} \in CC)$
$\mathbb{DS}$	$\{\mathbb{CC}, \mathbb{RN}, \mathbb{DS}\}$	$\times$
$\mathbb{DA}_{(j,i)}$ $\mathbb{DR}_{(j,i)}$	$\{\mathbb{CS}, \mathbb{CC}, \mathbb{RN}\}$	$\mathsf{conf}\{CC\} \wedge \langle cc, b^i \in B, T\rangle^{(j,u)} \in CC$
$\mathbb{CS}_j$	$\{\mathbb{CS}, \mathbb{CC}_{j'}, \mathbb{RN}_{(j',i)}\}$	$j \neq j'$
$\mathbb{CC}_j$	$\{\mathbb{CC}, \mathbb{RN}\}$	$j \neq j'$

Figure 5.42: At the top of the figure, we show an informal generic definition of the lemmas. The table shows, for each rule, the rules with which to commute and the constraints of the commutations.

a series of lemmas to provide a commutation relation between rule applications. In Table 5.42, we present this series of lemmas, showing on the first column the rule we want to commute, on the second column the set of rules commuting with it, and on the third column constraints if any. We formally present only the commutation lemmas for the `ServiceInit`, `ServiceStep`, and the device rules—`DeviceSensor`, `DeviceActuator`, and `DeviceReading`-. All the lemmas that concern the other rules follow the ones we describe.

First, we define the commutation lemma for the $\mathbb{IN}$ rule, as it is the rule repeatedly applied in the first execution step of the scheduler (Lemma 1).

Lemma 1 (Trace commutativity($\mathbb{IN}$)) *A `ServiceInit` transition relation can commute with all the other transitions. Commuting transitions results in the same final configuration.*

$$
\begin{aligned}
&\forall \mathsf{conf}, \mathsf{conf}', \mathsf{conf}'', \mathbb{X} : \\
&\mathbb{X} \in \{\mathbb{IN}, \mathbb{SS}, \mathbb{RS}, \mathbb{CS}, \mathbb{CC}, \mathbb{RN}, \mathbb{DV}\} \quad \wedge \quad \mathit{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'' \rightsquigarrow_{\mathbb{IN}} \mathsf{conf}' \\
&\Rightarrow \\
&\exists \mathsf{conf}''' : \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow_{\mathbb{IN}} \mathsf{conf}''' \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'
\end{aligned}
$$

We proved this lemma by case analysis on $\mathbb{X}$, showing that commuting the transition relation does not change the terminal configuration conf'. We show that, for each case we can commute the transition relation obtaining the same final configuration. Hence, repeatedly applying this lemma to a trace, batches all `ServiceInit`s present in the trace in the beginning. The transformation does not change the overall meaning of the trace. Indeed, the final configuration produced by executing `WEBI` following the reordered trace is equivalent to the original.

Similarly, we define a lemma for commuting the `ServiceStep` transition relation with a subset of $\mathbb{X}$. We present this commutation property in Lemma 2.

Lemma 2 (Trace commutativity($\mathbb{SS}$)) *Given* conf*, a `WEBI` configuration, and assuming limitations on the model described in the Hypotheses 1, 3, and 4:*

$$
\begin{aligned}
&\forall \mathsf{conf}, \mathsf{conf}', \mathsf{conf}'', \mathbb{X}, j, i, cc^{(j,u)}, b^i, CC \\
&\mathbb{X} \in \{\mathbb{SS}, \mathbb{RS}_{(j',i')}, \mathbb{CS}, \mathbb{CC}, \mathbb{RN}, \mathbb{DS}, \mathbb{DA}_{(j',i')}, \mathbb{DR}_{(j',i')}\} \quad \wedge \quad (j \neq j' \vee i \neq i') \quad \wedge \\
&\mathsf{conf}\{CC\} \quad \wedge \quad (\langle cc, b^i \in B, T\rangle^{(j,u)} \vee \langle \mathsf{boot}\rangle^{(j,u)} \in CC) \quad \wedge \\
&\mathit{wo}, \mathcal{E} \vdash \mathsf{conf}'' \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'' \rightsquigarrow_{\mathbb{SS}_{(j,i)}} \mathsf{conf}' \\
&\Rightarrow \\
&\exists \mathsf{conf}''' : \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow_{\mathbb{SS}_{(j,i)}} \mathsf{conf}''' \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'
\end{aligned}
$$

The lemma's proof is similar to the Lemma 1, obtaining the same result. There are three particular cases we want to discuss: commutation with another $\mathbb{SS}_{(j',i')}$—where $j \neq j' \wedge i \neq i'$-, with an $\mathbb{RS}_{(j,i)}$, and with the device rules. Regarding the commutation with another service, we know it is possible commute with the $\mathbb{SS}_{(j,i)}$ service, even if running on the same host. This is allowed because of the hypotheses regarding the

determinism of the server language and the untouched host memory. The $\mathbb{RS}_{(j,i)}$ never commutes with a future service step on the service (j, i). The reason is that the service (j, i) returns a value to the client. Afterward, it cannot step anymore. Hence, does not exist a trace that has a segment produced by the following sequence of transition relations: $\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}'' \rightsquigarrow_{\mathbb{RS}_{(j,i)}} \mathsf{conf}'' \rightsquigarrow_{\mathbb{SS}_{(j,i)}} \mathsf{conf}'$. Regarding devices, the service step can commute only with the device sensor transition, and the device actuator or reading signed by $j \neq j' \wedge i \neq i'$. Allowing commutation with an actuator or reading application of the same service means concretely to swap the service's lines of code.

Finally, we present three lemmas, one for each device rule—$\mathbb{DS}$ (Lemma 3), $\mathbb{DA}$ (Lemma 4), and $\mathbb{DR}$ (Lemma 5).

Lemma 3 (Trace commutativity($\mathbb{DS}$)) *Given* conf*, a `WEBI` configuration:*

$$
\begin{aligned}
&\forall \mathbb{X}, \mathsf{conf}, \mathsf{conf}', \mathsf{conf}''.\\
&\mathbb{X} \in \{\mathbb{DS}, \mathbb{CS}, \mathbb{CC}, \mathbb{RN}\} \quad \wedge \quad \mathit{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'' \rightsquigarrow_{\mathbb{DS}} \mathsf{conf}'\\
&\Rightarrow\\
&\exists \mathsf{conf}''' : \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow_{\mathbb{DS}} \mathsf{conf}''' \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'
\end{aligned}
$$

Lemma 4 (Trace commutativity($\mathbb{DA}$)) *Given* conf*, a webi configuration*

$$
\begin{aligned}
&\forall \mathbb{X}, \mathsf{conf}, \mathsf{conf}', \mathsf{conf}''.\\
&\mathbb{X} \in \{\mathbb{CS}, \mathbb{CC}, \mathbb{RN}\} \quad \wedge \quad \mathsf{conf}\{CC\} \quad \wedge \quad \langle cc, b^i \in B, T\rangle^{(j,u)} \in CC \quad \wedge\\
&\mathit{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'' \rightsquigarrow_{\mathbb{DA}_{(j,i)}} \mathsf{conf}'\\
&\Rightarrow\\
&\exists \mathsf{conf}''' : \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow_{\mathbb{DA}_{(j,i)}} \mathsf{conf}''' \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'
\end{aligned}
$$

Lemma 5 (Trace commutativity($\mathbb{DR}$)) *Given* conf*, a webi configuration:*

$$
\begin{aligned}
&\forall \mathbb{X}, \mathsf{conf}, \mathsf{conf}', \mathsf{conf}''.\\
&\mathbb{X} \in \{\mathbb{CS}, \mathbb{CC}, \mathbb{RN}\} \quad \wedge \quad \mathsf{conf}\{CC\} \quad \wedge \quad \langle cc, b^i \in B, T\rangle^{(j,u)} \in CC \quad \wedge\\
&\mathit{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'' \rightsquigarrow_{\mathbb{DR}_{(j,i)}} \mathsf{conf}'\\
&\Rightarrow\\
&\exists \mathsf{conf}''' : \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow_{\mathbb{DR}_{(j,i)}} \mathsf{conf}''' \rightsquigarrow_{\mathbb{X}} \mathsf{conf}'
\end{aligned}
$$

Also for these three lemmas, the proof approach is similar to the $\mathbb{IN}$ and $\mathbb{SS}$ lemmas. Regarding the device sensor lemma, we allow commutation among $\mathbb{DS}$s, but not with $\mathbb{DA}$ and $\mathbb{DR}$. For an actuation or a reading, putting a $\mathbb{DS}$ before or after changes the meaning of the world record $\mathcal{L}$. For example, allowing a commutation with a $\mathbb{DR}_{(j,i)}$ would probably mean reading a different value from the device memory, as the `DeviceSensor` rule produces a physical event and updates some devices' memories. For Lemma 4 and Lemma 5, we permit commutations only with the following client-driven rules: $\mathbb{CS}$, $\mathbb{CC}$, and $\mathbb{RN}$.

5.3.2.2 Interval Lemmas

The lemmas presented in Section 5.3.2.1 can be seen as the definition of order relations between `WEBI` labeled transitions $\mathsf{wo}, \mathcal{E} \vdash (\rightsquigarrow_{\mathbb{X}} \; R \; \rightsquigarrow_{\mathbb{X}'})$, for some $\mathbb{X}$, $\mathbb{X}'$, wo, $\mathcal{E}$, and an order relation R. These relations are fundamental for reordering the nondeterministic trace, obtaining an equivalent reordered one. The intuition behind the design of these lemmas is to reorder the trace using a sorting-like algorithmic approach.

In this section, we show two lemmas, one for creating an interval of `ServiceInit`s (step 1), and one for creating an interval of device rule applications (step 4). The other lemmas—one for each scheduler step-follow the two we define in the section.

Concerning step 1, we present the proposition in Lemma 6. To provide some notation, $\Pi_1 :: \Pi_2$ represents the concatenation of two traces, $\rightsquigarrow^n$ denotes multiple `WEBI` execution steps, $\rightsquigarrow^k_\Pi$ denotes k `WEBI` execution steps guided by the trace Π, and $|\Pi|$ represents the length of a trace.
We say that we can reorder a finite final trace Π_n, obtaining two traces, $\Pi^{\mathbb{IN}}$ and $\Pi^{\neg\mathbb{IN}}$, respectively, the trace of the $\mathbb{IN}$ rules application and the rest of the trace. A `WEBI` execution guided by the trace $\Pi^{\mathbb{IN}} :: \Pi^{\neg\mathbb{IN}}$ results in a final configuration equal to the one produced by executing the trace $\Pi^{\mathbb{IN}} :: \Pi^{\neg\mathbb{IN}}$.

Lemma 6 ($\mathbb{IN}$ Interval) *Given a well-formed initial `WEBI` configuration* conf *and a final*

configuration conf_n *produced after* n *steps of executing* `WEBI`*:*

$$\begin{gathered}
\forall \mathit{wo}, \mathcal{E}, \mathsf{conf}, \mathcal{I}, HM, IC, n, \mathsf{conf}_n, CC_e\mathcal{L}, IC', \Pi. \\
\mathcal{I} \neq \emptyset \quad \wedge \quad \mathsf{conf} = \langle [], \mathcal{I}, HM, \{\emptyset,\emptyset,\emptyset,\emptyset\}, \{\emptyset,\emptyset,\emptyset,\emptyset\}, IC, [] \rangle \\
\wedge \quad \mathit{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow^{n} \mathsf{conf}_n \quad \wedge \\
\mathsf{conf}_n = \langle \mathcal{L}, \emptyset, HM, \{\emptyset,\emptyset,\emptyset,\emptyset\}, \{\emptyset,\emptyset,\emptyset,CC_e\}, IC', \Pi \rangle \quad \wedge \quad |\mathcal{I}| = |CC_e| \\
\Rightarrow \\
\exists \mathsf{conf}_{k1}, \mathsf{conf}'_n, k1, k2, \Pi^{\mathbb{IN}}, \Pi^{\neg\mathbb{IN}}. \\
|\Pi^{\mathbb{IN}}| + |\Pi^{\neg\mathbb{IN}}| = k1 + k2 = n = |\Pi| \quad \wedge \quad |\Pi^{\neg\mathbb{IN}}| < |\Pi| \quad \wedge \\
\mathsf{conf}_{k1} = \langle [], \emptyset, HM, \{SS_r,\emptyset,\emptyset,\emptyset\}, \{CC_r,\emptyset,\emptyset,\emptyset\}, IC, \Pi^{\mathbb{IN}} \rangle \\
\wedge \quad SS_r \neq \emptyset \quad \wedge \quad CC_r \neq \emptyset \quad \wedge \quad \mathit{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow^{k1}_{\Pi^{\mathbb{IN}}} \mathsf{conf}_{k1} \rightsquigarrow^{k2}_{\Pi^{\neg\mathbb{IN}}} \mathsf{conf}'_n \quad \wedge \\
\mathsf{conf}'_n = \langle \mathcal{L}, \emptyset, HM, \{\emptyset,\emptyset,\emptyset,\emptyset\}, \{\emptyset,\emptyset,\emptyset,CC_e\}, IC', \Pi^{\mathbb{IN}} :: \Pi^{\neg\mathbb{IN}} \rangle
\end{gathered}$$

We proved Lemma 6 by strong induction on the length of the trace. We repeatedly apply Lemma 1 to every proof sub-case. Note that we deal with finite traces. Hence, we need a notion of reduction. On the right-hand side of the arrow, the condition $|\Pi^{\neg\mathbb{IN}}| < |\Pi|$ is satisfied, as reordering a portion of the trace makes the unordered part smaller than the original Π.

Regarding devices, we present the interval reordering in Lemma 7. We say that there always exists a reordering $\Pi^{\mathbb{DV}} :: \Pi^{\neg\mathfrak{S}'}$ of a non-ordered sub-trace $\Pi^{\neg\mathfrak{S}}$ such that an execution of a trace $\Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}} :: \Pi^{\neg\mathfrak{S}}$ results in a final `WEBI` configuration equivalent to the one produced by executing $\Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}} :: \Pi^{\mathbb{DV}} :: \Pi^{\neg\mathfrak{S}'}$.

The portion $\Pi^{\mathbb{DV}}$ of the trace is an interval of device rules.

Lemma 7 ($\mathbb{DV}$ **Interval)** *Given* conf*, an initial well-formed* `WEBI` *configuration, a finite portion of a trace* Π*,* $\Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}}$*, where* $\Pi^{\mathbb{IN}}$ *represents the step 1 reordering and* $\Pi^{\mathfrak{S}}$

some other reordering steps, $\Pi^{\neg\mathfrak{S}}$ *a finite non-reordered part of the trace:*

$$\begin{aligned}
&\forall \mathit{wo}, \mathcal{E}, k1, k2, n, \mathsf{conf}, \mathsf{conf}_{k1}, \mathsf{conf}_n, \mathcal{I}, HM, SS_a, SS_g,\\
&\qquad CC_r, CC_e, CC'_e, IC, IC', IC'', \mathcal{L}, \mathcal{L}', \Pi^{\mathbb{IN}}, \Pi^{\mathfrak{S}}, \Pi^{\neg\mathfrak{S}}.\\
&\qquad |\Pi^{\mathbb{IN}}| + |\Pi^{\mathfrak{S}}| = k1 \quad \wedge \quad k1 + |\Pi^{\neg\mathfrak{S}}| = n \quad \wedge\\
&\qquad \mathsf{conf} = \langle [], \mathcal{I}, HM, \{\emptyset,\emptyset,\emptyset,\emptyset\}, \{\emptyset,\emptyset,\emptyset,\emptyset\}, IC, [] \rangle \quad \wedge\\
&\qquad \mathit{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow^{k1}_{(\Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}})} \mathsf{conf}_{k1} \rightsquigarrow^{k2}_{\Pi^{\neg\mathfrak{S}}} \mathsf{conf}_n \quad \wedge\\
&\quad \mathsf{conf}_{k1} = \langle \mathcal{L}', \emptyset, HM, \{\emptyset, SS_a, SS_g, \emptyset\}, \{CC_r, \emptyset, \emptyset, CC'_e\}, IC'', \Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}} \rangle\\
&\wedge \quad \mathsf{conf}_n = \langle \mathcal{L}, \emptyset, HM, \{\emptyset,\emptyset,\emptyset,\emptyset\}, \{\emptyset,\emptyset,\emptyset, CC_e\}, IC', \Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}} :: \Pi^{\neg\mathfrak{S}} \rangle\\
&\qquad \wedge \quad |\mathcal{I}| = |CC_e| \quad \wedge \quad |CC'_e| < |\mathcal{I}|\\
&\qquad\qquad \Rightarrow\\
&\exists k1', k2', \mathsf{conf}_{(k1+k1')}, \mathsf{conf}'_n, SS_r, SS'_a, SS'_g, IC''', \mathcal{L}'', \Pi^{(\mathbb{DV})}, \Pi^{\neg\mathfrak{S}'}.\\
&|\Pi^{\neg\mathfrak{S}}| = k2 = k1' + k2' = |\Pi^{(\mathbb{DV})}| + |\Pi^{\neg\mathfrak{S}'}| \quad \wedge \quad |\Pi^{\neg\mathfrak{S}'}| \leq |\Pi^{\neg\mathfrak{S}}| \quad \wedge\\
&\mathit{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow^{k1}_{(\Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}})} \mathsf{conf}_{k1} \rightsquigarrow^{k1'}_{\Pi^{(\mathbb{DV})}} \mathsf{conf}_{(k1+k1')} \rightsquigarrow^{k2'}_{\Pi^{\neg\mathfrak{S}'}} \mathsf{conf}'_n \quad \wedge\\
&\mathsf{conf}_{(k1+k1')} = \langle \mathcal{L}'', \emptyset, HM, \{SS_r, SS'_a, SS'_g, \emptyset\}, \{CC_r, \emptyset, \emptyset, CC'_e\}, IC''', \Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}} :: \Pi^{\mathbb{DV}} \rangle \quad \wedge\\
&\mathsf{conf}'_n = \langle \mathcal{L}, \emptyset, HM, \{\emptyset,\emptyset,\emptyset,\emptyset\}, \{\emptyset,\emptyset,\emptyset, CC_e\}, IC', \Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}} :: \Pi^{\mathbb{DV}} :: \Pi^{\neg\mathfrak{S}'} \rangle
\end{aligned}$$

We use Lemmas 3, 4, and 5 to prove this lemma for applying commutations on the non-ordered trace $\Pi^{\neg\mathfrak{S}}$, resulting in a trace with a segment of $\mathbb{DV}$ rules (step 4), equivalent to the original—producing the same final `WEBI` configuration. We proceed by strong induction on the trace $\Pi^{\neg\mathfrak{S}}$. We can always create an interval of $\mathbb{DV}$s when SS_g or SS_a is not empty. Note that this interval could be empty if no device commutation lemma applies. In this case, the trace segment $\Pi^{\neg\mathfrak{S}}$ does not reduce. The postcondition $|\Pi^{\neg\mathfrak{S}'}| \leq |\Pi^{\neg\mathfrak{S}}|$ reflects this case.

5.3.2.3 Proof Sketch of Theorem 1

$\Leftarrow$ This direction is trivial. The non-deterministic `WEBI` semantics can always imitate a scheduler's step. Indeed, if the scheduler semantics can step, it is possible to do a step with non-deterministic `WEBI`.

$\Rightarrow$ We divide the proof into two steps. The first step reorders a non-deterministic trace by applying the commutation lemmas presented in Section 5.3.2.2 to cre-

ate a sequence of intervals in the trace. We show that the trace Π_n and the re-arranged (partially-ordered) trace Π'_n produce equivalent executions. We prove this step by defining two trace reordering iterations: one for the first iteration of the scheduler, comprehending the `ServiceInit`s, and the other for the successive iterations, which do not consider the scheduler's first step. For each sub-case, the final configuration is equivalent to the one obtained by executing the `WEBI` on the original one. Hence, the trace transformation is correct for the non-deterministic `WEBI` semantics. The proof comes almost in an algorithmic fashion. Despite its size, this proof step comes straightforward, as we rely on the commutation and the interval lemmas. The initial hypotheses we consider are the following.

$$\begin{aligned}
&\mathsf{wo}, \mathcal{E} \vdash \mathsf{conf} \rightsquigarrow^n \mathsf{conf}_n \\
&\mathsf{conf} = \langle [], \mathcal{I}, HM, \{\emptyset, \emptyset, \emptyset, \emptyset\}, \{\emptyset, \emptyset, \emptyset, \emptyset\}, IC, [] \rangle \\
&\mathsf{conf}_n = \langle \mathcal{L}, [], HM, \{\emptyset, \emptyset, \emptyset, \emptyset\}, \{\emptyset, \emptyset, \emptyset, CC_e\}, IC', \Pi_n \rangle
\end{aligned}$$

Moreover, the hypotheses we consider are the one listed in Section 5.3.1.1.

For the second step, we consider an ordered trace equivalent to Π_n, called $\Pi'_n = \Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}}$. Knowing that they produce equivalent executions, we show that both the scheduler and `WEBI` can simulate the ordered trace. We rewrite the theorem in the following proposition.

$$\begin{aligned}
&\forall\, \mathsf{wo}, \mathcal{E}, n, \Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}}, \mathsf{conf}_n, \mathsf{conf}'_n.\ \exists\, t : \\
&\qquad\qquad \mathsf{wo}, \mathcal{E} \vdash \mathsf{conf}\{[]\} \rightsquigarrow^n \mathsf{conf}_n\{\Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}}\} \\
&\qquad\qquad\qquad\qquad \Rightarrow \\
&(\mathsf{wo}, \mathcal{E} \vdash [\![\mathsf{conf}]\!]_{\mathbb{IN}}^{\pi_{\mathfrak{L}}(\Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}}), \pi_{\mathfrak{T}}(\Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}})} \rightsquigarrow_{\mathfrak{S}}^{(n+t)} [\![\mathsf{conf}'_{(n)}]\!]_{\mathtt{end}}^{[],[]}) \ \wedge\ \mathsf{conf}'_n = \mathsf{conf}_n
\end{aligned}$$

For simplicity, we rename $\Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}}$ as $\Pi'^{\mathfrak{S}}$, where $|\Pi'^{\mathfrak{S}}| = n$. If the trace size n is 0, the proof is trivial. Otherwise, we define an induction predicate stating that the scheduler can always imitate the reordered trace for some generic $k + t$ steps, where t is the number of scheduler transitions that do not change the

configuration. We show the induction predicate in the following proposition:

$$\forall n, k, \Pi'^{\mathfrak{S}}, \Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}}, \mathsf{conf}, \mathsf{conf}_k.$$
$$\Pi^{\mathbb{IN}} :: \Pi^{\mathfrak{S}} = \Pi'^{\mathfrak{S}} \quad \wedge \quad |\Pi'^{\mathfrak{S}}| = n \quad \wedge \quad \mathsf{conf} \rightsquigarrow^{k}_{\Pi'^{\mathfrak{S}}_{[0,k]}} \mathsf{conf}_k$$
$$\Rightarrow$$
$$\exists t, \mathbb{X}.$$
$$[\![\mathsf{conf}]\!]^{\pi_{\mathfrak{L}}(\Pi'^{\mathfrak{S}}),\pi_{\mathfrak{T}}(\Pi'^{\mathfrak{S}})}_{\mathbb{IN}} \rightsquigarrow^{(k+t)}_{\mathfrak{S}} [\![\mathsf{conf}'_k]\!]^{\pi_{\mathfrak{L}}(\Pi'^{\mathfrak{S}}_{[k+1,n]}),\pi_{\mathfrak{T}}(\Pi'^{\mathfrak{S}}_{[k+1,n]})}_{\mathbb{X}} \quad \wedge$$
$$\mathsf{conf}'_k = \mathsf{conf}_k$$

, where $\Pi'^{\mathfrak{S}}_{[0,k]}$ is the subtrace of $\Pi'^{\mathfrak{S}}$ from the index 0 to k.

We assume that this predicate hold for k steps. We show that it also holds for every next step $k+1$. We go by case analysis on k, showing that the scheduler and the non-deterministic semantics can do $k + t$ and k steps respectively, following the trace $\Pi'^{\mathfrak{S}}_{[0,k]}$.
Then, for each possible sub-case, we show that both the semantics can perform one more execution step to produce the same $k + 1$ configuration.
The induction predicate holds for every subcase; hence we conclude the proof by saying that the scheduler's execution can produce every final configuration generated by a `WEBI` execution, making the semantics equivalent.

5.3.3 Executing the Example of Section 5.2.3 in Skel

In the section, we show how the implementation of the scheduler in Skel is suitable for executing the example presented in Section 5.2.3.

Skel Implementation. The scheduler semantics we proposed is one designed for proving the equivalence theorem. To implement the scheduler, we could have implemented it by providing as input $\mathfrak{L}$ and $\mathfrak{T}$. However, the usability of the scheduler would then be complex. Thus, we relax the input that should be provided and implement a scheduler such that :

- Scheduling step 4 does not follow the trace $\mathfrak{L}$.

- Scheduling step 7 does not follow the trace $\mathfrak{T}$.

To observe the possible interaction in the scheduler's execution, we design steps 4 and 7 to be non-deterministic, similarly to diagram B in Figure 5.33. Indeed, we exploit the Skel `branch` construct for deciding whether to pick and execute configurations with elements from SS_a, SS_g, or CC_t, or not.

All the other steps are immuted, emptying the related partition set before passing to the next scheduling step.

We derive two interpreters from the Skel definition, one using the `RandID` interpretation monad and the other using the `List` monad. One can find the examples in the following repository: `https://gitlab.inria.fr/skeletons/webi-in-skel/-/tree/master`.

Two Scheduler Executions. We showed that reducing the non-determinism of the model does not affect the expressiveness of `WEBI`. The OCaml interpreter generated from the scheduler's semantics produces all the possible traces generated by the original semantics interpreter's executions. This section presents two traces equivalent to traces A and B described in Figure 5.32. The result of executing these traces produces final `WEBI` configurations congruent to the ones presented in Section 5.2.3. We remark that the interpreter follows the behavior of the diagram shown in Figure 5.33.

Figure 5.43 presents the equivalent traces resulting from a concrete interpretation. These two traces perform four scheduler iterations.

The first iteration starts on both cases with two `ServiceInit`s, and the order of the initializations is not essential. Afterward, both the services perform `ServiceSteps`, choosing to do two steps on a row on the `windowManager` and one on the `tunOnOven` service. The resulting configuration contains the oven actuation and the thermometer reading in the partition sets `_SS_a` and `_SS_g`. These devices' interactions belong to the execution of the `windowManager` and the `tunOnOven` running services. The first iteration ends with the device rules applications. Trace A first executes the device actuator of the oven after a device sensor, meaning that the following `DeviceSensor` will update the thermometer memory to $26°$, making the `DeviceReading` reading a temperature higher than $24°$C. Differently, trace B first executes the `DeviceSensor` and the `DeviceReading`, making the read temperature being a value lower than $24°$C.

The second iteration is substantially different between traces A and B. Trace A first makes the window and the oven service code progress via `ServiceSteps`. This

Scheduler's execution equivalent to rule sequence A.

1st Iteration.
`ServiceInit` Step
$\rightarrow \mathbb{IN} \rightarrow \mathbb{IN}$
`ServiceStep` Step
$\rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_o$
`DevRules` Step
$\rightarrow \mathbb{DS} \rightarrow \mathbb{DA}_o \rightarrow \mathbb{DS} \rightarrow \mathbb{DR}_w$
2nd Iteration.
`ServiceStep` Step
$\rightarrow \mathbb{SS}_o \rightarrow \mathbb{SS}_o \rightarrow \mathbb{SS}_w$
$\rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w$
$\rightarrow \mathbb{SS}_w$
`RetService` Step
$\rightarrow \mathbb{RS}_o$
`DevRules` Step
$\rightarrow \mathbb{DS} \rightarrow \mathbb{DA}_w$
`ClientStep` Step
$\rightarrow \mathbb{CS}_s$
3rd Iteration.
`ServiceStep` Step
$\rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w$
`DevRules` Step
$\mathbb{DS} \rightarrow \mathbb{DR}_w$
4th Iteration.
`ServiceStep` Step
$\rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w$
`RetService` Step
$\rightarrow \mathbb{RS}_w$
`ClientStep` Step
$\rightarrow \mathbb{CS}_a$

Scheduler's execution equivalent to rule sequence B.

1st Iteration.
`ServiceInit` Step
$\rightarrow \mathbb{IN} \rightarrow \mathbb{IN}$
`ServiceStep` Step
$\rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_o$
`DevRules` Step
$\rightarrow \mathbb{DS} \rightarrow \mathbb{DR}_w$
2nd Iteration.
`ServiceStep` Step
$\rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w$
$\rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w$
`DevRules` Step
$\rightarrow \mathbb{DS} \rightarrow \mathbb{DA}_w \rightarrow \mathbb{DS} \rightarrow \mathbb{DA}_o$
3rd Iteration.
`ServiceStep` Step
$\rightarrow \mathbb{SS}_o \rightarrow \mathbb{SS}_o \rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w$
$\rightarrow \mathbb{SS}_w$
`RetService` Step
$\rightarrow \mathbb{RS}_o$
`DevRules` Step
$\rightarrow \mathbb{DS} \rightarrow \mathbb{DR}_w$
`ClientStep` Step
$\rightarrow \mathbb{CS}_s$
4th Iteration.
`ServiceStep` Step
$\rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w \rightarrow \mathbb{SS}_w$
`RetService` Step
$\rightarrow \mathbb{RS}_w$
`ClientStep` Step
$\rightarrow \mathbb{CS}_a$

Figure 5.43: Two scheduler's reordered traces behaviorally equivalent to the traces in Figure 5.32. With equivalent we mean that they produce the same final `WEBI` configuration. We annotate the some rules applications with "o" and "w" to refer to the `oven` and `window` service, and "a" and "s" to refer to the clients `adam` and `steve`.

series of `ServiceStep`s concludes the oven service evaluation in trace A. Indeed, in the next step, a `RetServiceBoot` happens, creating a running client. Afterward, a device actuator happens for the `windowManager` service, and finally, the client is created by the oven service steps, concluding the second iteration. For trace B, things work differently.

Only the `windowManager` steps because the oven actuation is still pending. Afterward, a series of device rules happen by performing an actuation—and closing it-and turning on the oven after a device sensor application. Trace B does not produce a configuration with a detectable interaction between the two devices.

The two traces also differ in the third iteration. Trace A performs some service steps on the `windowManager` service before reaching the thermometer `get`. Afterward, the trace records a device sensor followed by the reading of the only service executing. Trace B completes the execution of the `turnOnOven` service and continues the one of the `windowManager`—similarly to trace A. It follows that the window service reads the thermometer temperature, and finally oven's service returns, creating a running client, ending this iteration.

The fourth iteration of the scheduler is identical for both traces, concluding the execution of the window service, which by returning creates a running client, which with a client step ends its execution too.

Collecting Semantics. Execution traces A and B produce the only two final configurations the model can derive. We executed the same program with the collecting semantics monad, obtaining different permutations of the traces, meaning that device rule applications are located differently in the resulting $\mathfrak{L}$. Nevertheless, the two final configurations are always equivalent to the ones produced by executing the traces in Figure 5.43.

Conclusion

In this thesis, we defined a practical approach to represent the semantics of programming languages via Skel [41], a specification language that concretizes Skeletal Semantics [8]. In Chapter 3, we have shown how this simple, purely functional meta-language can easily capture complex target language features by leveraging monadic operators. We claim the descriptions written in Skel are easy to read and maintain. Indeed, new meta features (such as the need to manipulate continuations) are seldom introduced, and when they are, only the monadic binders need to be adapted; existing descriptions not relying on the feature may be left unchanged.
In addition, existing tools such as `necroml` [42] let us derive an OCaml interpreter from a Skeletal Semantics. We have used it to obtain executable versions of the toy languages presented in this chapter—the arithmetic language and its extension with exceptions, the PCF, the stateful PCF, and the yield language.[5]

In Chapter 4, we have described how can the Skel language be used to mechanize complex semantics. In particular, we have shown how carefully chosen monads can help to have a formal description close to the specification while still describing the complex behaviors of ES. This work applied on a larger scale the technique presented in Chapter 3. We decided to formalize JavaScript because of two main reasons. First, we have previous experience dealing with the ES specification [6]. Second, the ES specification is both very precise and very complex. Hence we do not have to guess the behavior of the language while making sure Skel can scale. In addition to implementing parts of ES in Skel, we also provide boilerplate code to integrate with existing JS parsers and implement all the unspecified terms in OCaml. Using `necroml`, we generate an OCaml executable semantics.
Our main effort stood in mechanizing the foundations of the language, of which `GetValue` is a good example. We can now easily extend our formalization by following the same systematic approach for other abstract algorithms and language constructs.
We have also evaluated the maintainability of the approach. This project began with the formalization of ECMAScript 2020, but after a few months, we switched to the newer ECMAScript 2021. This process took only a few days of work. Naturally, this will change once we complete the formalization of the standard. Nevertheless, we can produce a list of differences between specifications (as a textual diff), making the amount of work proportional to the extent of changes and not to the specification's size. In particular, unlike other work [6], we only have one mechanization to change instead of many. The

5. `https://gitlab.inria.fr/akhayam/programming_skel`

standard changes quite often, not only by modifying algorithmic steps but also by introducing side-effects into the specification.
The work of this chapter is a real-world language application of the work of Chapter 3). We showed that it is possible to systematically introduce side-effects into a formalization by delegating all the necessary machinery to handle them to some carefully designed monads. It is a well-known result [72], but it is unclear how it could be applied in the context of Skel. The monadic approach lets us introduce a new behavior throughout the specification by simply changing the monad we use. Indeed, a short-term goal is to implement a new monad for manipulating delimited continuations and combining it with the ones presented in this paper. As discussed in Section 4.1.6, we will then be able to implement generators whose execution can be suspended (using `yield`) and resumed. This feature is necessary to implement function calls as its algorithmic description uses a generator. Once function calls are implemented, we can directly run the ECMA-262 test suite.
In the long run, we plan several developments for this work. The first goal is to validate that we can use our mechanized semantics to prove some properties of the language. One property of interest is the guarantee that assertions in the specification are always satisfied. To this end, we will use the `necrocoq` tool [50]. As the formalization of JS helped us refine the Skel language, we believe the generation of a Coq semantics will provide many opportunities to suggest improvements for `necrocoq`. Another goal is to exploit our systematic approach to produce a Skeletal Semantics directly from the ES document, using a tool similar to the one presented by *J. Park et al.* [59]. Finally, we plan to apply our expertise to formalize other complex languages in Skel. An example could be Python, for which formal semantics has been described on paper [45].

In Chapter 5, we studied Skel from another perspective. We wanted to see if the language is suitable for designing a non-deterministic model. The model describes the behavior of distributed systems, which consider connected devices, for writing IoT applications. Hence, after explaining `WEBI` extensively, we provided a refined small-step operational semantics of this pre-existing model and its Skel implementation.
From this implementation, we derived two interpreters on which we showed and executed examples of programs. The first instantiation of the interpreter evaluates programs returning a result, and the second instantiation collects all the possible execution traces. However, the collecting interpretation was not executable, as the traces to collect on the naive implementation of `WEBI` are infinite, making the interpretation run out

of memory.
For this reason, we tamed the non-determinism intrinsic in this model by designing a scheduled semantics of WEBI equivalent to the original under certain assumptions. We showed that from a non-deterministic trace, there is always a trace transformation to build an equivalent scheduler trace for each execution trace. This equivalence result is vital for enabling preliminary studies on this model. Indeed, at the end of the chapter, we showed that we executed both the concrete and the collecting semantics interpreter with the scheduler semantics without having any memory issues. Especially the collecting semantics is interesting, as it paves the way for analysis of the execution of programs on WEBI. It is a significant result, as the model, passing its embryonal phase, is ready for further formal studies, both in program analysis and programming language semantics. Indeed, a plan for the future is to use a server and a client variant of the ECMAScript semantics proposed in Chapter 4 to provide formal semantics to a multi-tier language, such as Hop.js [65]. Thus, we would be able to derive a certified version of the language, which relies on the certifications of the clients and the service languages, and write and analyze IoT applications that we will define as secure concerning some security properties. The path is long, but this thesis provides modular work that can be merged for formally reasoning on IoT applications. As said, the two directions are to design scalable static analysis techniques on the Skel version of WEBI and to complete the ECMAScript formalizations for developing a multi-tier language we will use for writing programs. We will rely on the correctness of the work of Chapter 4 once it is finished.

Final Considerations. The approach presented in Chapter 3 and the semantics of JavaScript described in Chapter 4 pushed the Skel language to evolve. The Skel evolution and the works presented in the thesis went in parallel, showing how a more applicative version of a theoretical framework can lead to the growth of a language that was more of an academic tool for representing toy languages. This consideration is positive, as the goal was to have a language and a set of tools reflecting a theoretical framework usable by programming language designers and programmers. We did not change the essence of the original Skeletal Semantics framework [8], edulcorating it for being intuitive to use for real language formalizations. We claim that our technique leads to building solid formalizations. Moreover, at the time of the writing, we have stable tools for deriving a concrete interpretation [42].

However, the `necrocoq` tool is not yet ready. Once `necrocoq` is complete, Skel will be a powerful tool for encoding and certifying programming language semantics.

The original paper states a result of consistency regarding different interpretations, namely the concrete and the abstract interpretation. Skel's tool environment is evolving to include derivations of annotated artefacts for performing abstract interpretation, making the development of such tool essential to transform Skel into a complete language for semanticists.

Finally, the work done for `WEBI` is introductory. Still, we have all the intents to use the Skel tools to study this model for studying IoT security properties, an actual research problem, as these devices are increasingly part of our everyday life. The aim is to be able to explore IoT application's security via instantiations of the Skel semantics of `WEBI`.

I am confident that Skeletal Semantics and Skel are the perfect cost-effective formal framework and specification language for reaching these goals.

BIBLIOGRAPHY

[1] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt, "Certified Abstract Machines for Skeletal Semantics", *in*: *Certified Programs and Proofs*, Philadelphia, United States, Jan. 2022.

[2] *APACHE Groovy, A multi-faceted language for the Java platform*, URL: https://groovy-lang.org.

[3] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and P. Naur, "Revised Report on the Algorithm Language ALGOL 60", *in*: *Commun. ACM* 6.*1* (1963), pp. 1–17, ISSN: 0001-0782, DOI: 10.1145/366193.366201, URL: https://doi.org/10.1145/366193.366201.

[4] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy, "Formal Verification of a C Compiler Front-End", *in*: *FM 2006: Int. Symp. on Formal Methods*, vol. 4085, Lecture Notes in Computer Science, Springer, 2006, pp. 460–475, URL: http://xavierleroy.org/publi/cfront.pdf.

[5] Sandrine Blazy and Xavier Leroy, "Formal verification of a memory model for C-like imperative languages", *in*: *International Conference on Formal Engineering Methods (ICFEM 2005)*, vol. 3785, Lecture Notes in Computer Science, Springer, 2005, pp. 280–299, URL: http://xavierleroy.org/publi/memory-model.pdf.

[6] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith, "A Trusted Mechanised JavaScript Specification", *in*: *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 49 (Jan. 2014), pp. 87–100, DOI: 10.1145/2578855.2535876.

[7] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith, *JSCert: Certified JavaScript*, URL: https://jscert.org.

[8] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt, "Skeletal Semantics and their Interpretations", *in*: *Proceedings of the ACM on Programming Languages* 44 (2019), pp. 1–31, DOI: 10.1145/3290357, URL: https://hal.inria.fr/hal-01881863.

[9] Martin Bodin, Thomas Jensen, and Alan Schmitt, "Certified Abstract Interpretation with Pretty-Big-Step Semantics", *in*: *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, Mumbai, India: Association for Computing Machinery, 2015, pp. 29–40, ISBN: 9781450332965, DOI: 10.1145/2676724.2693174, URL: https://doi.org/10.1145/2676724.2693174.

[10] Denis Bogdanas and Grigore Roşu, "K-Java: A Complete Semantics of Java", *in*: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, Mumbai, India: Association for Computing Machinery, 2015, pp. 445–456, ISBN: 9781450333009, DOI: 10.1145/2676726.2676982, URL: https://doi.org/10.1145/2676726.2676982.

[11] Gérard Boudol, Zhengqin Luo, Tamara Rezk, and Manuel Serrano, "Reasoning about Web Applications: An Operational Semantics for HOP", *in*: *ACM Trans. Program. Lang. Syst.* 34.*2* (June 2012), ISSN: 0164-0925, DOI: 10.1145/2220365.2220369, URL: https://doi.org/10.1145/2220365.2220369.

[12] *CAKEML: A Verified Implementation of ML*, URL: https://cakeml.org.

[13] Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac, "Sensitive Information Tracking in Commodity IoT", *in*: *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, Baltimore, MD, USA: USENIX Association, 2018, pp. 1687–1704, ISBN: 9781931971461.

[14] Z. Berkay Celik, Patrick McDaniel, and Gang Tan, "SOTERIA: Automated IoT Safety and Security Analysis", *in*: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, Boston, MA, USA: USENIX Association, 2018, pp. 147–158, ISBN: 9781931971447.

[15] Arthur Charguéraud, "Pretty-big-step semantics", *in*: *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*, Springer, 2013, pp. 41–60.

[16] Arthur Charguéraud, Alan Schmitt, and Thomas Wood, "JSExplain: A Double Debugger for JavaScript", *in*: *The Web Conference 2018*, Lyon, France, 2018, pp. 1–9, DOI: 10.1145/3184558.3185969.

[17] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu, "Towards a Trustworthy Semantics-Based Language Framework via Proof Generation", *in*: *Computer Aided Verification*, ed. by Alexandra Silva and K. Rustan M. Leino, Cham: Springer International Publishing, 2021, pp. 477–499, ISBN: 978-3-030-81688-9.

[18] *COMPCERT*, URL: https://compcert.org.

[19] *Coq Program Extraction*, URL: https://coq.inria.fr/refman/addendum/extraction.html.

[20] ECMA, *ECMAScript 2021 Language Specification*, 2021, URL: https://tc39.es/ecma262/2021/.

[21] Chucky Ellison and Grigore Rosu, "An Executable Formal Semantics of C with Applications", *in*: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, Philadelphia, PA, USA: Association for Computing Machinery, 2012, pp. 533–544, ISBN: 9781450310833, DOI: 10.1145/2103656.2103719, URL: https://doi.org/10.1145/2103656.2103719.

[22] *engine262, An implementation of ECMA-262 in JavaScript*, URL: https://github.com/engine262/engine262.

[23] Daniele Filaretti and Sergio Maffeis, "An Executable Formal Semantics of PHP", *in*: *ECOOP 2014 – Object-Oriented Programming*, ed. by Richard Jones, Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 567–592, ISBN: 978-3-662-44202-9.

[24] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen, "The Essence of Compiling with Continuations", *in*: *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, ed. by Robert Cartwright, ACM, 1993, pp. 237–247, ISBN: 0-89791-598-4, DOI: 10.1145/155090.155113, URL: https://doi.org/10.1145/155090.155113.

[25] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi, "The Essence of JavaScript", *in*: *ECOOP, Lecture Notes in Computer Science* (June 2010), pp. 126–150, DOI: 10.1007/978-3-642-14107-2_7.

[26] John Hannan and Dale Miller, "From Operational Semantics to Abstract Machines: Preliminary Results", *in*: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, Nice, France: Association for Computing Machinery, 1990, pp. 323–332, ISBN: 089791368X, DOI: 10.1145/91556.91680, URL: https://doi.org/10.1145/91556.91680.

[27] Robert Harper, Furio Honsell, and Gordon Plotkin, "A Framework for Defining Logics", *in*: *J. ACM* 40.*1* (1993), pp. 143–184, ISSN: 0004-5411, DOI: 10.1145/138027.138060, URL: https://doi.org/10.1145/138027.138060.

[28] Chris Hathhorn, Chucky Ellison, and Grigore Roşu, "Defining the Undefinedness of C", *in*: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, Portland, OR, USA: Association for Computing Machinery, 2015, pp. 336–345, ISBN: 9781450334686, DOI: 10.1145/2737924.2737979, URL: https://doi.org/10.1145/2737924.2737979.

[29] *HOL*, URL: https://hol-theorem-prover.org/.

[30] *Isabelle*, URL: https://isabelle.in.tum.de.

[31] Roshan P. James, A. U.S., and A. Sabry, "Yield: Mainstream Delimited Continuations", *in*: 2011.

[32] Gilles Kahn, "Natural Semantics", *in*: *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '87, Berlin, Heidelberg: Springer-Verlag, 1987, pp. 22–39, ISBN: 354017219X.

[33] Adam Khayam, Louis Noizet, and Alan Schmitt, "A Faithful Description of ECMAScript Algorithms", *in*: *Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming*, PPDP '22, Tbilisi, Georgia: Association for Computing Machinery, 2022, ISBN: 9781450397032, DOI: 10.1145/3551357.3551381, URL: https://doi.org/10.1145/3551357.3551381.

[34] Adam Khayam, Louis Noizet, and Alan Schmitt, "JSkel: Towards a Formalization of JavaScript's Semantics", *in*: *JFLA 2021 - Journées Francophones des Langages Applicatifs*, Virtuelles, France, Apr. 2021, pp. 1–22, URL: https://hal.inria.fr/hal-03509431.

[35] Adam Khayam and Alan Schmitt, *A Practical Approach for Describing Language Semantics*, Submitted for publication - The Art, Science, and Engineering of Programming, May 2022.

[36] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler, "Run Your Research: On the Effectiveness of Lightweight Mechanization", *in*: *SIGPLAN Not.* 47.*1* (2012), pp. 285–296, ISSN: 0362-1340, DOI: 10.1145/2103621.2103691, URL: https://doi.org/10.1145/2103621.2103691.

[37] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens, "CakeML: A Verified Implementation of ML", *in*: *Principles of Programming Languages (POPL)*, ACM Press, Jan. 2014, pp. 179–191, DOI: 10.1145/2535838.2535841, URL: https://cakeml.org/popl14.pdf.

[38] Peter J. Landin, "The Mechanical Evaluation of Expressions", *in*: *Comput. J.* 6 (1964), pp. 308–320.

[39] Xavier Leroy, "Formal Verification of a Realistic Compiler", *in*: *Commun. ACM* 52.*7* (July 2009), pp. 107–115, ISSN: 0001-0782, DOI: 10.1145/1538788.1538814, URL: https://doi.org/10.1145/1538788.1538814.

[40] Liyi Li and Elsa L. Gunter, "A Complete Semantics of K and Its Translation to Isabelle", *in*: *Theoretical Aspects of Computing – ICTAC 2021: 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8–10, 2021, Proceedings*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 152–171, ISBN: 978-3-030-85314-3, DOI: 10.1007/978-3-030-85315-0_10, URL: https://doi.org/10.1007/978-3-030-85315-0_10.

[41] Noizet Louis and Schmitt Alan, "Semantics in Skel and Necro", *in*: *Italian Conference on Theoretical Computer Science, Rome, Italy, September 7-9, 2022*, 2022.

[42] Enzo Crance Martin Bodin Nathanael Courant and Louis Noizet, *Necro Ocaml Generator, https://gitlab.inria.fr/skeletons/necro-ml*, URL: https://gitlab.inria.fr/skeletons/necro-ml.

[43] Robert Milne and C. Strachey, *A Theory of Programming Language Semantics*, 99th, USA: Halsted Press, 1977, ISBN: 0470989068.

[44] Robin Milner, Mads Tofte, and David Macqueen, *The Definition of Standard ML*, Cambridge, MA, USA: MIT Press, 1997, ISBN: 0262631814.

[45] Raphaël Monat, "Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries", PhD thesis, Sorbonne University, 2021.

[46] Peter D Mosses, "Modular structural operational semantics", *in*: *The Journal of Logic and Algebraic Programming* 60-61 (2004), Structural Operational Semantics, pp. 195–228, ISSN: 1567-8326, DOI: https://doi.org/10.1016/j.jlap.2004.03.008, URL: https://www.sciencedirect.com/science/article/pii/S156783260400027X.

[47] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell, "Lem: Reusable Engineering of Real-World Semantics", *in*: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 175–188, ISBN: 9781450328739, DOI: 10.1145/2628136.2628143, URL: https://doi.org/10.1145/2628136.2628143.

[48] Magnus O. Myreen and Scott Owens, "Proof-Producing Synthesis of ML from Higher-Order Logic", *in*: *SIGPLAN Not.* 47.9 (2012), pp. 115–126, ISSN: 0362-1340, DOI: 10.1145/2398856.2364545, URL: https://doi.org/10.1145/2398856.2364545.

[49] Louis Noizet, *Necro Debugger Generator, https://gitlab.inria.fr/skeletons/necro-debug*, URL: https://gitlab.inria.fr/skeletons/necro-debug.

[50] Louis Noizet, *Necro Gallina Generator, https://gitlab.inria.fr/skeletons/necro-coq*, URL: https://gitlab.inria.fr/skeletons/necro-coq.

[51] Louis Noizet, *Necro Library, https://gitlab.inria.fr/skeletons/necro*, URL: https://gitlab.inria.fr/skeletons/necro.

[52] Louis Noizet and Alan Schmitt, "Semantics in Skel and Necro", *in*: *ICTCS 2022 - Italian Conference on Theoretical Computer Science*, CEUR Workshop Proceedings, Rome, Italy, Sept. 2022, pp. 1–17, URL: https://hal.inria.fr/hal-03784478.

[53] Louis Noizet and Alan Schmitt, *Stating and Handling Semantics with Skel and Necro*, Research Report RR-9449, Inria Rennes - Bretagne Atlantique, Jan. 2022, pp. 1–23, URL: https://hal.inria.fr/hal-03543701.

[54] Louis Noizet and Alan Schmitt, *Stating and Handling Semantics with Skel and Necro*, Research Report, Inria Rennes - Bretagne Atlantique, Jan. 2022, p. 20, URL: https://hal.inria.fr/hal-03543701.

[55] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell, "Lem: A Lightweight Tool for Heavyweight Semantics", *in*: *Interactive Theorem Proving*, ed. by Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 363–369, ISBN: 978-3-642-22863-6.

[56] Daejun Park, Andrei Stefănescu, and Grigore Roşu, "KJS: A Complete Formal Semantics of JavaScript", *in*: *ACM SIGPLAN Notices* 50 (June 2015), pp. 346–356, DOI: 10.1145/2813885.2737991.

[57] Jihyeok Park, Seungmin An, and Sukyoung Ryu, "Automatically Deriving JavaScript Static Analyzers from Specifications Using Meta-Level Static Analysis", *in*: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 1022–1034, ISBN: 9781450394130, DOI: 10.1145/3540250.3549097, URL: https://doi.org/10.1145/3540250.3549097.

[58] Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu, "JSTAR: JavaScript Specification Type Analyzer using Refinement", *in*: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 606–616, DOI: 10.1109/ASE51524.2021.9678781.

[59] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu, "JISET: JavaScript IR-based Semantics Extraction Toolchain", *in*: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 647–658.

[60] Blaise Pascal, *The Provincial Letters*, 1657.

[61] Gordon D Plotkin, *A structural approach to operational semantics*, Aarhus university, 1981.

[62] Gordon D. Plotkin, "LCF Considered as a Programming Language", *in*: *Theoretical Computer Science* 5.*3* (1977), pp. 223–255, DOI: 10.1016/0304-3975(77)90044-5, URL: https://doi.org/10.1016/0304-3975(77)90044-5.

[63] Grigore Roşu and Traian Şerbănuţă, "An Overview of the K Semantic Framework", *in*: *The Journal of Logic and Algebraic Programming* 79 (Aug. 2010), pp. 397–434, DOI: 10.1016/j.jlap.2010.03.012.

[64] Amr Sabry and Matthias Felleisen, "Reasoning About Programs in Continuation-Passing Style", *in*: *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*, ed. by Jon L. White, ACM, 1992, pp. 288–298, ISBN: 0-89791-481-3, DOI: 10.1145/141471.141563, URL: https://doi.org/10.1145/141471.141563.

[65] Manuel Serrano, *Hop, multitier Web Programming*, 2006, URL: http://hop.inria.fr.

[66] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša, "Ott: Effective Tool Support for the Working Semanticist", *in*: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, Freiburg, Germany: Association for Computing Machinery, 2007, pp. 1–12, ISBN: 9781595938152, DOI: 10.1145/1291151.1291155, URL: https://doi.org/10.1145/1291151.1291155.

[67] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy, "Retrofitting Effect Handlers onto OCaml", *in*: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, Virtual, Canada: Association for Computing Machinery, 2021, pp. 206–221, ISBN: 9781450383912, DOI: 10.1145/3453483.3454039, URL: https://doi.org/10.1145/3453483.3454039.

[68] *Standard ML Family GitHub Project*, URL: https://smlfamily.github.io.

[69] Yong Kiam Tan, Scott Owens, and Ramana Kumar, "A Verified Type System for CakeML", *in*: IFL '15, Koblenz, Germany: Association for Computing Machinery, 2015, ISBN: 9781450342735, DOI: 10.1145/2897336.2897344, URL: https://doi.org/10.1145/2897336.2897344.

[70] *The Coq proof assistant*, URL: https://coq.inria.fr.

[71] *The Twelf Project*, URL: http://twelf.org/wiki/Main_Page.

[72] Philip Wadler, "The Essence of Functional Programming", *in*: *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, 1992, pp. 1–14, DOI: 10.1145/143165.143169, URL: https://doi.org/10.1145/143165.143169.

[73] C.P. Wadsworth, *Semantics and Pragmatics of the Lambda-calculus*, University of Oxford, 1971.

[74] Qi Wang, Wajih Hassan, Adam Bates, and Carl Gunter, "Fear and Logging in the Internet of Things", *in*: Jan. 2018, DOI: 10.14722/ndss.2018.23291.